SOFTWARE





DIENSTPROGRAMME
BAND A

EAW *electronic*

P8000

Version 1.2 (2008-03-02)

W E G A - S o f t w a r e

Dienstprogramme

(Band-A)

Diese Dokumentation wurde von einem Kollektiv des Kombinates

VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"

erarbeitet.

Herausgeber:
Kombinat
VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"
Hoffmannstrasse 15-26
BERLIN
1193

WAE/03-0202-03

Ausgabe: 12/87

Die vorliegende Dokumentation unterliegt nicht dem Aenderungsdienst.

Spezielle Hinweise zum aktuellen Stand der Softwarepakete befinden sich in README-Dateien auf den entsprechenden Vertriebsdisketten.

Dieser Band-A enthaelt folgende Unterlagen:

Teil 1: C-SHELL
Eine Einfuehrung in C-Shell
(Bearbeiter: U. Oefler)

Teil 2: SHELL
Eine Einfuehrung in Shell
(Bearbeiter: U. Oefler)

Teil 3: EX/VI
Kurzbeschreibung der Editoren EX und VI
(Bearbeiter: U. Oefler)

Teil 4: COMM
WEGA-Kommunikationspaket
(Bearbeiter: L. Mielenz)

Teil 5: U8000-PLZ/ASM
Benutzerhandbuch
(Bearbeiter: U. Oefler)

Teil 6: U8000-PLZ/SYS
Benutzerhandbuch
(Bearbeiter: J. Zabel)

Teil 7: U8000
Aufrufvereinbarungen
(Bearbeiter: U. Oefler)

Teil 8: MAKE
Programmbeschreibung
(Bearbeiter: R. Ruppelt)

Gegenueber der vorherigen Ausgabe wurde der Teil 4 (COMM) ueberarbeitet.

Weitere Dokumentationen zu WEGA-Dienstprogrammen werden z.Z. erarbeitet. Sie erscheinen nach Fertigstellung in Folgebaenden.

Gesamtinhaltsverzeichnis Seite

Teil 1: CSHELL

Teil 2: SHELL

Teil 3: EX/VI

Teil 4: COMM

Teil 5: U8000 PLZ/ASM

Teil 6: U8000 PLZ/SYS

Teil 7: U8000 Aufrufvereinbarungen

Teil 8: MAKE

Hinweise des Lesers zu diesem Dokumentationsband

Wir sind staendig bemueht, unsere Unterlagen auf einem qualitativ hochwertigen Stand zu halten. Sollten Sie deshalb Hinweise zur Verbesserung dieses Dokumentationsbandes bzw. zur Beseitigung von Fehlern haben, so bitten wir Sie, diesen Fragebogen auszufuellen und an uns zurueckzusenden.

Titel des Dokumentationsbandes: WEGA-Dienstprogramme (Band-A)

Ihr Name / Tel.-Nr.:

Name und Anschrift des Betriebes:

Genuegt diese Dokumentation Ihren Anspruechen? ja / nein
Falls nein, warum nicht?

Was wuerde diese Dokumentation verbessern?

Sonstige Hinweise:

Fehler innerhalb dieser Dokumentation:

Unsere Anschrift: Kombinat VEB ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW "FRIEDRICH EBERT"
Abteilung Basissoftware
Hoffmannstrasse 15-26
BERLIN
1193

C S H E L L

Eine Einfuehrung in C-Shell

Vorwort

Diese Beschreibung erlaeutert die Moeglichkeiten von C-Shell und gibt Hinweise zum Gebrauch. Sie stellt eine erste Einfuehrung in C-Shell dar.
In Ergaenzung hierzu wird auf die Dokumentation csh(1) des WEGA-Programmierhandbuchs (Online-Dokumentation) verwiesen.

C-Shell ist ein Kommandointerpreter unter dem System WEGA. Sie bietet zur normalen WEGA-Shell einige zusaetzliche erweiterte Moeglichkeiten fuer die interaktive Arbeit (z.B. die History-Funktion). C-Shell besitzt einer der Programmiersprache C angelehnte Syntax.

Inhaltsverzeichnis Seite

1. Interaktive Benutzung von C-Shell

1.1. Kommandos

Unter WEGA ist eine Shell ("Schale des Betriebssystems") hauptsaechlich ein Medium, mit deren Hilfe Kommandos aufgerufen werden. C-Shell besitzt eine Menge eingebauter (interner) Kommandos, die direkt abgearbeitet werden koennen. Die meisten allgemeinen Kommandos sind jedoch extern. Was C-Shell von Kommandointerpretern anderer Systeme unterscheidet, ist der Fakt, dass es ein Anwenderprogramm darstellt, das fast ausschliesslich dafuer da ist, andere Programme aufzurufen.

Die Kommandos im System WEGA erwarten eine Liste von Zeichenketten oder Worten als Argumente. Zum Beispiel besteht das Kommando

```
mail bill
```

aus zwei Worten. Das erste Wort 'mail' bezeichnet das auszufuehrende Kommando (in diesem Fall ein Kommando, das eine Nachricht an einen anderen Nutzer uebermittelt). C-Shell sucht in einer Anzahl von Directorys nach der Datei 'mail', die das Programm enthaelt. Der Rest der Kommandozeile wird dem Kommando zur Auswertung uebergeben. In diesem Fall wird das Wort 'bill' vom Programm 'mail' als der Name eines Nutzers interpretiert, an den die Nachricht zu uebermitteln ist. Das Kommando 'mail' wird normalerweise folgendermassen verwendet

```
%mail bill
Ich habe eine Frage zur CSH-Dokumentation.
Bei mir fehlt die 5. Seite.
Existiert sie bei Dir?
                        Peter
%
```

Eine Nachricht wird an den Nutzer 'bill' gesendet. Die Eingabe der Nachricht muss mit CTRL-d, das eine EOF-Nachricht fuer das Programm 'mail' darstellt, beendet werden. Das sogenannte Promptzeichen '%' wird vor und nach dem Kommando 'mail' ausgegeben, um anzuzeigen dass C-Shell zur Kommandoeingabe bereit ist.
Nach Ausgabe des Promptzeichens liest C-Shell die Kommandoeingaben vom Terminal ein. Nach Eingabe von 'mail bill' fuehrt C-Shell das Kommando 'mail' mit dem Argument 'bill' aus und wartet auf das Ende der Ausfuehrung. Das Programm 'mail' liest bis zum Erkennen von EOF Eingaben vom Terminal ein und teilt C-Shell dann mit, dass es fertig ist. C-Shell signalisiert dem Nutzer durch Ausgabe des Promptzeichens, dass es wieder fuer Terminaleingaben bereit ist.

Das ist das Grundmuster der Zusammenarbeit zwischen WEGA und C-Shell. Ein vollstaendiges Kommando wird vom Terminal

aus eingegeben, C-Shell fuehrt es aus und zeigt nach Abschluss der Ausfuehrung die Eingabebereitschaft durch Ausgabe des Promptzeichens an. Diese Vorgehensweise wird durch die Abarbeitungszeit der Kommandos nicht beeinflusst. Wenn z.B. der Editor ueber eine Stunde laeuft, so wartet C-Shell solange, bis der Editorlauf beendet wird und gibt erst dann sein Promptzeichen wieder aus.

1.2. Flags

Flagargumente beginnen normalerweise mit einem Minuszeichen (-) und spezifizieren optionale Moeglichkeiten der Kommandoabarbeitung. Zum Beispiel erzeugt das Kommando

```
ls
```

eine Liste aller Dateien des aktuellen Directorys. Wenn die Groessenoption (-s) dazugefuegt wird,

```
ls -s
```

so erfolgt zusaetzlich die Groessenangabe in Bloecken von 512 Byte fuer jede Datei. Im WEGA-Programmierhandbuch sind fuer jedes Kommando alle verfuegbaren Optionen angegegeben.

1.3. Ausgabe an Dateien

Viele Kommandos arbeiten bei der Ein-/Ausgabe mit Dateien anstelle des Terminals. Beim Aufruf der Kommandos kann durch eine besondere Notation angezeigt werden, wohin der Ausgabestrom gehen soll. Dies erfolgt bei C-Shell bevor die Kommandos ausgefuehrt werden. Das Kommando

```
date
```

gibt das aktuelle Systemdatum ueber das Terminal aus. Das Terminal stellt implizit die Standardausgabe fuer das Kommando 'date' dar. Um das aktuelle Datum in einer Datei zu retten, ist eine Neuzuweisung des Standard-Ausgabestroms moeglich. Dies erfolgt durch Angabe des Metazeichens '>' gefolgt von einem Dateinamen. Somit erzeugt das Kommando

```
date > now
```

eine Datei 'now', die das aktuelle Systemdatum enthaelt. Dabei ist wichtig zu bemerken, dass das Kommando 'date' durch die Neuzuweisung des Ausgabestroms nicht veraendert wird. C-Shell organisiert diese Umlenkung vor der Kommandoausfuehrung.

Die Datei 'now' braucht vor Ausfuehrung des Kommandos 'date' nicht zu existieren; C-Shell erzeugt sie in einem solchen Fall. Wenn sie bereits existiert, so wird ihr Inhalt ueberschrieben. Ueber die C-Shell-Option

'noclobber' (s. Abschn. 2.2.) kann ein Ueberschreiben von Dateiinhalten verhindert werden.

1.4. Metazeichen in C-Shell

C-Shell besitzt eine Reihe von speziellen Zeichen (wie >), die eine besondere Funktion anzeigen. Im allgemeinen haben die Zeichen, die keine Buchstaben oder Ziffern sind, eine spezielle syntaktische oder semantische Bedeutung fuer C-Shell. Metazeichen wirken normalerweise, wenn C-Shell Eingaben liest.

Metazeichen koennen nicht direkt als Teile von Woertern eingesetzt werden. Zum Beispiel gibt das Kommando

```
echo *
```

nicht den Stern aus. Es gibt entweder eine sortierte Liste aller Dateinamen des aktuellen Directorys aus, oder es erfolgt die Ausgabe der Nachricht "No match", falls keine Dateien in dem aktuellen Directory vorhanden sind.

Der empfohlene Mechanismus fuer den Gebrauch von Metazeichen als Argumente ist, sie in Apostrophe (') einzuschliessen, z.B.

```
echo '*'
```

Ein Zeichen, das Ausrufungszeichen (!) (von der C-Shell-History-Funktion benutzt) kann nicht auf diese Weise behandelt werden.
Das Ausrufungszeichen (!) und das Apostroph (') verlieren ihre besondere Bedeutung, wenn ihnen ein Backslashzeichen (\) vorangestellt wird. Diese beiden Mechanismen reichen aus, um jedes druckbare Zeichen in einem Wort, das ein Argument eines C-Shell-Kommandos ist, unterzubringen.

1.5. Eingabe von Dateien

Die Standardeingabe fuer ein Kommando kann auch von einer Datei kommen. Eine entsprechende Zuweisung des Eingabestroms ist fuer eine Vielzahl von Kommandos nicht notwendig, da sie als Argumente angegebene Dateien verarbeiten. So wird durch

```
sort < data
```

das Kommando 'sort' gestartet, wobei die Standardeingabe von der Datei 'data' kommt. Es ist einfacher

```
sort data
```

einzugeben. Dann eroeffnet das Kommando 'sort' die Datei 'data' selber zur Eingabe. Wenn nur

```
        sort
```

eingegeben wird, dann sortiert das Programm alle von der Standardeingabe kommenden Zeilen. Da keine Neuzuweisung des Eingabestroms erfolgte, werden alle vom Terminal eingegebenen Zeilen bis zum Erkennen von CTRL-d sortiert.

Pipeline

C-Shell kann die Standardausgabe eines Kommandos mit der Standardeingabe eines anderen Kommandos verbinden. Diese Vorgehensweise wird als Pipeline bezeichnet. Kommandos, zwischen denen ein senkrechter Strich (|) steht, werden durch C-Shell miteinander verknuepft, d.h., der Ausgabestrom des einen Kommandos wird mit dem Eingabestrom des anderen Kommandos verbunden. Das Kommando

```
        ls -s
```

erzeugt z.B. eine Liste aller Dateien des aktuellen Directorys einschliesslich ihrer Groesse in Bloecken von 512 Zeichen. Durch Kombinieren des Kommandos mit Optionen des Kommandos 'sort' koennen die Dateinamen nach anderen Kriterien anstelle der alphabetischen Reihenfolge sortiert werden. Die Option '-n' legt fuer das Kommando 'sort' ein Sortieren nach der Groesse fest. Unter Zuhilfename einer Pipeline kann durch

```
        ls -s | sort -n
```

eine nach der Dateigroesse sortierte Liste (die kleinste zuerst) erzeugt werden. Durch Angabe der Option '-r' bei 'sort' kann die Sortierreihenfolge umgedreht werden. In Kombination mit dem Kommando 'head' werden durch

```
        ls -s | sort -n -r | head -5
```

die Dateinamen einchliesslich Groesse der fuenf groessten Dateien des aktuellen Directorys ausgegeben.

1.6. Dateinamen

Jede WEGA-Datei besitzt einen Dateinamen (max. 14 Zeichen lang). Jede Datei wird durch einen Namenseintrag in einem Directory repraesentiert. Die Verbindung von Dateien und Directorys wird durch Pfadnamen ausgedrueckt. WEGA-Pfadnamen bestehen aus einer Anzahl von Komponenten, die durch einen Schraegstrich (/) getrennt werden. Jede Komponente mit Aussnahme der letzten stellt einen Directorynamen dar, in der die naechste Komponente (wieder Directory- oder Dateiname) enthalten ist. Zum Beispiel spezifiziert der Pfadname

/etc/motd

die Datei 'motd' im Directory 'etc', welches ein Unterdirectory des Wurzeldirectorys (/) darstellt. Pfadnamen, die nicht mit einem Schraegstrich beginnen, werden ab dem aktuellen Directory beginnend interpretiert. Dieses Directory ist implizit das Home-Directory des Nutzers. Es kann natuerlich per Kommando (chdir oder cd) auch ein anderes Directory das aktuelle sein.

Alle druckbaren Zeichen mit Ausnahme vom Schraegstrich (/) koennen in Dateinamen auftauchen. Zeichen mit einer besonderen Bedeutung sollten aber vermieden werden. Die meisten Dateinamen bestehen aus einer Anzahl von alphanumerischen Zeichen und Punkten (.). Der Punkt ist kein Metazeichen und wird deshalb oft zur Trennung oder Erweiterung von einem Basisnamen benutzt, z.B. sind

prog.c prog.o prog.errs prog.output

vier Dateinamen, die einen gemeinsamen Beginn und unterschiedliche Erweiterungen besitzen. Die Datei 'prog.c' ist ein C-Quellprogramm, die Datei 'prog.o' die entsprechende Objektdatei, die Datei 'prog.errs' enthaelt eine Liste von Uebersetzungsfehlern und die Datei 'prog.output' wurde vom uebersetzten Programm erzeugt.
Die Metanotation

prog.*

kann in einem Kommando benutzt werden, um auf alle Dateien zu verweisen, die mit 'prog.' beginnen. Dieses Wort wird von Shell (bevor die Kommandoausfuehrung beginnt) in eine Liste von Dateinamen, die mit 'prog.' beginnen umgewandelt. Der Stern (*) steht dabei fuer eine beliebige Zeichenkette (einschliesslich der leeren) im Dateinamen. Die Namen, die in Frage kommen, werden als alphabetisch sortierte Liste dem Kommando als Argumente uebergeben. Das Kommando

echo prog.*

gibt in unserem Beispiel

prog.c prog.errs prog.o prog.output

aus. Das Kommando 'echo' erhaelt vier Woerter als Argumente, die durch die Dateinamensaufloesung von 'prog.*' durch Shell erzeugt werden.

Ein anderes zur Dateinamenserzeugung verwendbares Zeichen ist das Fragezeichen (?). Es steht fuer genau ein beliebiges Zeichen im Dateinamen. Zum Beispiel gibt

echo ? ?? ???

eine Liste von Dateinamen aus; zuerst die, die nur aus

einem Zeichen bestehen, danach die aus zwei Zeichen und zum Schluss die aus drei Zeichen. Die Dateinamen werden pro Groesse alphabetisch sortiert.

Ein anderer Mechanismus besteht aus der Angabe einer Folge von Zeichen zwischen eckigen Klammern ([]). Diese Metasequenz steht dann fuer jedes einzelne Zeichen aus der eingeschlossenen Menge. Zum Beispiel steht

```
prog.[co]
```

in unserem erwaehnten Beispiel fuer

```
prog.c   prog.o
```

Zwei durch einen Bindestrich (-) getrennte Zeichen stehen fuer den Bereich, z.B. steht

```
chap.[1-5]
```

fuer die Dateien

```
chap.1   chap.2   chap.3   chap.4   chap.5
```

(falls sie existieren). Dies ist eine abkuerzende Schreibweise fuer

```
chap.[12345]
```

und ist zu ihr aequivalent.

Falls bei der Aufloesung von Metanotationen fuer Dateinamen keine Uebereinstimmung mit existierenden Dateinamen festgestellt wird, so erfolgt von Shell die Ausgabe der Fehlernachricht

```
No match.
```

Der Punkt als erstes Zeichen eines Dateinamens wird besonders behandelt. Die Abkuerzungsnotationen fuer Dateinamen *, ? und [] wirken fuer den Punkt nicht. Das verhindert eine Uebereinstimmung mit Dateinamen, die fuer das System eine spezielle Bedeutung haben (wie . und ..) und fuer Dateien, die normalerweise unsichtbar sind (wie .cshrc).

Ein weiterer Mechanismus gibt die Moeglichkeit, den Pfadnamen des Homedirectorys eines anderen Nutzers zu erzeugen. Die entsprechende Notation besteht aus der Tilde (~) gefolgt von einem Nutzer-Login-Namen. Zum Beispiel steht das Wort ~bill fuer den Pfadnamen /z/bill, falls dies sein Home-Directory ist. Vorallem in groesseren Systemen koennen die Nutzer-Login-Directorys aus einer Folge von mehreren Directorynamen bestehen. Dann ist die Verwendung der eben erwaehnten Notation fuer den Zugriff auf Dateien von anderen Nutzern guenstig.

Ein spezieller Fall ist das Tildezeichen allein, z.B. ~/mbox. Es steht fuer das eigene Home-Directory des Nutzers. Dies kann sehr guenstig sein, wenn der Anwender sich in anderen Directorys befindet und sich entschliesst, Dateien in sein Home-Directory zu kopieren. Der Aufruf

```
cp thatfile ~
```

wird durch C-Shell in

```
cp thatfile /z/bill
```

aufgeloest, wenn /z/bill das Home-Directory des Nutzers ist.

Anders als die Metanotationen (*, ? und []) bewirkt die Tilde nicht den Test ob, angegebene Dateien existieren. Dies ist z.B. beim Gebrauch des Kommandos 'cp' guenstig, wie in

```
cp thatfile ~/saveit
```

Geschweifte Klammern koennen zur Abkuerzung einer Menge von Woertern, die gemeinsame Teile haben, benutzt werden. Die eben erwaehnten Abkuerzungsmechanismen koennen dabei nicht benutzt werden, da es sich dabei nicht um Dateinamen, sondern um Zeichenketten handelt. Dieser Mechanismus wird im Abschn. 4.2. beschrieben.

1.7. Abbruch von Kommandos

Es ist moeglich, ein laufendes Programm abzubrechen, waehrend C-Shell untaetig ist, ohne C-Shell selber zu beenden. Wenn z.B. das Kommando

```
cat /etc/passwd
```

eingegeben wurde, so erfolgt die Ausgabe der Liste aller Nutzer des Systems ueber das Terminal. Durch druecken der DEL- oder RUB-Taste wird ein Interruptsignal an alle auf dem Terminal laufenden Programme, einschliesslich C-Shell, gesendet. C-Shell ignoriert normalerweise diese Signale. So ist es nur das Programm 'cat', das durch diesen Interrupt beeinflusst wird. Es besitzt keinen Mechanismus, um den Interrupt zu ignorieren. Nach Beendigung des Programms verlaesst C-Shell seinen Wartezustand und gibt das Promptzeichen (%) aus. Falls ein weiterer Interrupt auftritt, so wiederholt C-Shell die Promptzeichenausgabe, da es Interruptsignale ignoriert.

Viele Programme werden beim Empfangen einer EOF-Nachricht von der Standardeingabe abgebrochen. Das Programmbeispiel 'mail' aus Abschn. 1.1. wird bei Eingabe von CTRL-d (entspricht EOF) beendet. C-Shell wird ebenso beendet, wenn es ein EOF empfaengt. WEGA schaltet den Anwender dann

aus dem System aus (logout). Das bedeutet, dass die Eingabe von zu vielen CTRL-d's zu einem Ausschalten aus dem System fuehren kann. C-Shell bietet aber einen Mechanismus dieses zu verhindern. Dazu dient die im Abschn. 2.2. beschriebene Option 'ignoreeof'.

Kommandos, die ihre Standardeingabe von einer Datei erhalten werden normalerweise bei Erreichen des Dateiendes beendet. So wird

```
mail bill < prepared.text
```

bei Lesen des Dateiendes (d.h. EOF) von 'prepared.text' beendet.

Programme, die noch nicht vollstaendig ausgetestet sind, koennen durch Eingabe von CTRL-\ gestoppt werden. C-Shell reagiert mit einer Nachricht:

```
a.out: Quit -- Core dumped
```

Dies zeigt an, dass eine Datei 'core' erzeugt wurde, die Informationen ueber den Status des abgebrochenen Programms 'a.out' enthaelt. Falls Programme im Hintergrund laufen, so ignorieren sie vom Terminal eingegebene Interrupt- und Quit-Signale. Um solche Programme anzuhalten ist das Kommando 'kill' anzuwenden (s. Abschn. 2.6. fuer ein Beispiel).

2. Details der C-Shell-Operationen

2.1. Start und Beendigung

Beim Einschalten eines Nutzers (login) wird vom System C-Shell gestartet, und es wird begonnen, Kommandos aus der Datei '.cshrc' (falls im Home-Directory des Nutzers vorhanden) einzulesen. Bei jedem Start von allen weiteren C-Shell-Prozessen werden die in der Datei '.cshrc' enthaltenen Kommandos ausgefuehrt.
Bei jedem Login des Nutzers werden die in der Datei '.login' enthaltenen Kommandos ausgefuehrt. Das folgende Beispiel ist ein typisches Beispiel fuer eine Login-Datei

```
setenv TERM adm3a
set history=20
set time=3
```

Das erste Kommando (setenv) informiert das System, dass der Nutzer mit einem ADM3A-Terminal angeschlossen ist. Die naechsten zwei Kommandos 'set' werden direkt von C-Shell interpretiert. Sie setzen die Werte von zwei Variablen fuer C-Shell und beeinflussen dadurch die weitere Arbeitsweise von C-Shell.
Das Setzen der Variablen 'time' teilt C-Shell mit, dass es fuer Kommandos, die eine bestimmte Abarbeitungszeit

ueberschreiten (in diesem Fall drei CPU-Sekunden) eine Zeitstatistik auszugeben hat.
Das Setzen der Variablen 'history' teilt C-Shell die Tiefe der zu speichernden Kommandos mit. In diesem Fall werden immer die 20 zuletzt ausgefuehrten Kommandos gespeichert. Der Anwender kann mit Hilfe der History-Funktion vorherige Kommandos wiederholen oder etwas modifiziert wieder starten. C-Shell setzt diese Variable nicht implizit. Nutzer, die diese Funktion nutzen wollen, muessen diese Variable setzen. Der Wert 20 ist ein relativ grosser Wert dafuer, i.allg. arbeitet man mit Groessen zwischen 5 und 10. Der Gebrauch der History-Funktion wird im Abschn. 2.3. beschrieben.

Nach Ausfuehrung der Kommandos aus der Datei '.login' zeigt C-Shell die Eingabebereitschaft durch Ausgabe des Promptzeichens (%) an. Bei der Eingabe von EOF (CTRL-D) vom Terminal gibt C-Shell die Nachricht 'logout' aus und fuehrt die in der Datei '.logout' (falls im Home-Directory vorhanden) enthaltenen Kommandos aus. Danach wird C-Shell beendet und das System schaltet den Nutzer vom System ab (logout).

2.2. C-Shell-Variablen

C-Shell arbeitet mit einer Reihe von Variablen, die als Wert ein Feld von leeren oder mehreren Zeichenketten haben. Diesen Variablen koennen durch das Kommando 'set' Werte zugewiesen werden. Die gebraeuchlichste Form ist

```
set name=wert
```

C-Shell-Variablen koennen auch zum Speichern von Werten benutzt werden, die nach Ausfuehrung des Substitutionsmechanismus an Kommandos uebergeben werden sollen. Die meisten globalen C-Shell-Variablen beziehen sich auf C-Shell selber. Durch Aendern der Werte dieser Variablen ist es moeglich, direkt die Arbeitsweise von C-Shell zu beeinflussen.

Eine der wichtigen Variablen ist 'path', die eine Folge von Directorynamen enthaelt, in denen C-Shell nach Kommandos sucht. Das Kommando 'set' ohne Argumente zeigt die Werte fuer alle augenblicklich in C-Shell definierten Variablen.

```
%set
argv    ()
home    /z/bill
path    (. /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
```

Die Notation zeigt an, dass die Variable 'path' auf das aktuelle Directory (.), dann auf /bin und schliesslich noch

auf /usr/bin verweist. Vom Nutzer geschriebene Kommandos koennen in '.' (gewoehnlich ein Directory des Nutzers) vorhanden sein. Die meisten wichtigen Systemkommandos liegen in /bin, zusaetzlich sind noch weitere Kommandos in /usr/bin zu finden.

Eine nuetzliche interne Variable ist 'home', die das Nutzer-Home-Directory festlegt.

Die Variable 'ignoreeof' kann in der Datei '.login' gesetzt werden. Dadurch wird verhindert, dass bei Eingabe von EOF (CTRL-D) C-Shell verlassen wird. Um sich dann aus WEGA auszuschalten ist die Eingabe von

```
        logout
```

notwendig. Die Variable 'ignoreeof' wird durch

```
        set ignoreeof
```

gestzt und durch

```
        unset ignoreeof
```

zurueckgesetzt.
Beide Kommandos 'set' und 'unset' sind interne Kommandos von C-Shell.

Eine weitere interne C-Shell-Variable ist 'noclobber', die das Ueberschreiben von Dateien verhindert, wenn sie gesetzt ist. Durch

```
        >filename
```

wird der Inhalt der angegebenen Datei, wenn sie bereits existiert, durch Ueberschreiben zerstoert. Um das zu verhindern ist

```
        set noclobber
```

(z.B. in der Datei '.login' untergebracht) einzugeben. In diesem Fall wird bei jeder Zuweisung

```
        >filename
```

erst getestet, ob die Datei bereits existiert. Durch die spezielle Schreibweise '>!' ist aber auch in einem solchen Fall ein Ueberschreiben moeglich.

```
z.B.    date >! now
```

Die Variable 'mail' ist ebenso intern. Um sicher zu gehen, dass eingehende Post angezeigt wird, waehrend man eingeschaltet ist, muss die Variable 'mail' gesetzt sein, z.B. durch

```
set mail=/z/mail/yourname
```

C-Shell ueberprueft dann alle 10 Minuten, ob Post angekommen ist. Da das Setzen dieser Variablen die Antwortzeiten von C-Shell verzoegern kann, sollte sie nur bei haeufigen Postverkehr im System gesetzt sein.

Der Gebrauch von C-Shell-Variablen und Zeichenketten in Kommandos, was haeufig in Kommandoskripten notwendig ist, wird im Abschn. 2.4. erlaeutert.

2.3. History-Liste

C-Shell verarbeitet eine History-Liste, die die Kommandozeilen der zuletzt ausgefuehrten Kommandos enthaelt. Durch Angabe einer speziellen Notation ist es moeglich, Kommandos oder Woerter davon erneut aufzurufen, oder daraus neue Kommandos zu bilden oder aber Tippfehler zu korrigieren. Die folgenden Beispiele verdeutlichen diese Arbeitsweise.

```
%where michael
michael is on tty0  dialup 300baud 642-792
%write !$
write michael
Lange Zeit nicht gesehen.
Warum rufst du mich nicht an?
EOF
%
```

Das System gibt als Antwort auf 'where michael' aus, dass er am Kanal tty0 angeschlossen ist. Danach wird das Kommando 'write' mit dem Argument '!$' aufgerufen. Dies ist eine speziell History-Notation und kennzeichnet das letzte Wort des zuletzt ausgefuehrten Kommandos.
Die folgende Kommandofolge koennte moeglich sein, wenn keine Reaktion von Michael erfolgt.

```
%ps -t0
  PID  TTY  TIME  COMMAND
 4808   0   0:05  -
%!!
ps -t0
  PID  TTY  TIME  COMMAND
 5104   0   0:00  - 7
%!where
where michael
michael is not logged in
%
```

Das Kommando 'ps' zeigt an, dass er auf dem Kanal tty0 eingeschaltet ist, aber keine Shell darauf laeuft. Dieses Kommando wird durch die History-Notation '!!' erneut gestartet und zeigt nun an, dass er sich abgeschaltet hat. Die Wiederholung des Kommandos 'where' zeigt an, dass er

sich wirklich abgeschaltet hat.

Die Form '!!' wiederholt die letzte Kommandoausfuehrung. Die Form '!string' wiederholt das letzte Kommando, das die angegebene Zeichenkette zu Beginn enthaelt.

Eine weitere zur Korrektur von Tippfehlern nuetzliche Kommandoform ist '^alt^neu', die eine zu 'ed' oder 'ex' aehnliche Substitution ausfuehrt.

```
%cat ~bill/csh/sh..c
/mnt/bill/csh/sh..c: No such file or directory
%^..^.
cat ~bill/csh/sh.c

            .
    Ausgabe der Datei
            .

%
```

In diesem Beispiel werden durch die Notation '^..^.' im letzten Kommando die beiden Punkte durch einen Punkt ersetzt und das Kommando erneut gestartet.

Das folgende Kommando kann zur Ausgabe der Datei ueber den Drucker benutzt werden.

```
%!! | lpr
cat ~bill/csh/sh.c | lpr
```

Direkt nach Ausfuehrung des Kommandos 'cat' kann auch unter Verwendung des Kommandos 'pr' eine Ausgabe ueber den Drucker erfolgen.

```
%pr !$ | lpr
pr ~bill/csh/sh.c | lpr
```

Darueber hinaus bietet die History-Funktion noch weitergehende Moeglichkeiten. Ein Beispiel dafuer ist

```
%cd !$:h
cd ~bill/csh
```

Das angehaengte ':h' bewirkt, dass nur der erste Teil des Pfadnamens durch den History-Mechanismus substituiert wird. Dieser Mechanismus und andere Formen werden weniger als die hier beschriebenen Moeglichkeiten benutzt.

Eine komplette Beschreibung der Moeglichkeiten der History-Funktion wird unter csh(1) im WEGA-Programmierhandbuch gegeben.

2.4. Alias-Funktion

C-Shell besitzt eine sogenannte Alias-Funktion, die eine Transformation von eingegebenen Kommandos durchfuehrt. Sie ist der Makromoeglichkeit von einigen Assemblern aehnlich. Einige der durch alias bereitgestellten Moeglichkeiten koennen auch unter Verwendung von C-Shell-Kommandodateien realisiert werden. Dadurch koennen aber nicht die augenblickliche Umgebung von C-Shell und die Kommandos selber beeinflusst werden.

Ein Beispiel dazu. Vorausgesetzt es existiert eine neue Version des Kommandos 'mail', das an Stelle der alten Version benutzt werden soll. Durch Angabe des C-Shell-Kommandos

```
alias mail Mail
```

in der Datei '.login' des Nutzers wird eine Eingabezeile der Form

```
mail bill
```

in einen Aufruf von 'Mail' transformiert. Um beim Aufruf von 'ls' die Ausgabe der Dateigroesse mit zu veranlassen ist

```
alias ls ls -s
```

moeglich. Durch

```
alias dir ls -s
```

wird ein neues Kommando 'dir' erzeugt, das 'ls -s' ausfuehrt. Die Eingabe

```
dir ~bill
```

wird von C-Shell in

```
ls -s /z/bill
```

umgewandelt.

Somit koennen durch den Alias-Mechanismus kurze Namen fuer Kommandos festgelegt, implizit Argumente fixiert und neue Namen fuer Kommandos ausgewaehlt werden. Ebenso ist es moeglich mehrere Kommandos oder eine Pipeline in der Alias-Funktion anzugeben. Die Definition

```
alias cd 'cd \!*;ls'
```

fuehrt das Kommando 'ls' nach dem Wechsel des Directorys (Kommando 'cd') aus. Die ganze Alias-Definition ist in Apostrophe eingeschlossen, um die meisten Substitutionen zu verhindern. Dem Anfuehrungszeichen wurde ein Backslash-

zeichen vorangestellt um zu verhindern, dass es bei der Eingabe des Alias-Kommandos interpretiert wird. Die Notation '\!*' substituiert die gesamte Argumentliste des durch Alias neu definierten Kommandos 'cd', ohne dass ein Fehler auftritt, wenn keine Argumente angegeben werden. Das Semikolon (;) trennt die Kommandos.

Durch

```
alias whois 'grep \!^ /etc/passwd'
```

wird ein Kommando definiert, das sein erstes Argument in der Passwort-Datei sucht.

2.5. Hintergrundkommandos und Ein-/Ausgabezuordnung

Das Metazeichen & kann nach einem Kommando oder einer durch Semikolon oder | getrennten Kommandofolge angegeben werden. Dadurch wird vermieden, dass C-Shell auf das Ende der Kommandoabarbeitung wartet. Diese Kommandos werden dann als Hintergrundkommandos oder -prozesse bezeichnet. Im folgenden Beispiel

```
%pr ~bill/csh/sh.c|lpr&
5120
5121
%
```

gibt C-Shell zwei Prozessnummern aus und kehrt sofort zum Eingabedialog zurueck, ohne auf das Ende der Ausfuehrung der Kommandos 'pr' und 'lpr' zu warten. Die Zahlen 5120 und 5121 sind die vom System den Kommandos 'pr' und 'lpr' zugeordneten Prozessnummern.
Eine Abarbeitung von Kommandos im Hintergrund belastet das System, d.h., die Nutzerantwortzeiten des Systems vergroessern sich, besonders wenn das System dadurch sehr stark belastet wird (d.h. sehr viele Prozesse gleichzeitig laufen).

Es koennen verschiedene Probleme auftreten, wenn ein Hintergrundkommando Eingaben vom Nutzerterminal liest, waehrend zur gleichen Zeit C-Shell Kommandos vom Terminal einliest. Um dies zu verhindern, ist die implizite Standardeingabe fuer Hintergrundkommandos nicht das Terminal, sondern eine leere Datei (/dev/null). Hintergrundkommandos werden durch Interrupt- und Quit-Signale, die vom Terminal erzeugt werden, nicht beeinflusst.

Wenn es notwendig ist sich abzuschalten (log off) bevor das Kommando beendet ist, so muss das Kommando das Signal 'hangup' ignorieren, d.h., das Kommando laeuft nach Ausschalten des Terminals weiter. Dies kann erreicht werden, indem vor dem Kommando 'nohup' angegeben wird.

```
nohup man csh | nohup lpr&
```

Zusaetzlich zur Standardausgabe besitzen Kommandos noch eine Fehlerausgabe (diagnostic output), die normalerweise auch dem Terminal zugeordnet ist, auch wenn die Standardausgabe einer Datei oder einer Pipe zugeordnet ist. Dieser Fehlerausgabestrom kann auch mit dem Standardausgabestrom verschmolzen werden. Wenn der Standardausgabestrom eines laenger laufenden Programms einer Datei zugewiesen wurde, koennte es nuetzlich sein auch die Fehlermeldungen dort abzulegen.
Durch die Notation

```
Kommando >& Datei
```

wird C-Shell mitgeteilt, dass sowohl der Fehlerausgabestrom als auch der Standardausgabestrom der angegebenen Datei zugeordnet ist. Aehnlich koennen durch

```
Kommando |& lpr
```

beide Ausgabestroeme durch eine Pipeline ueber das Kommando 'lpr' an den Drucker ausgegeben werden. Die folgende Form

```
Kommando >&! Datei
```

ist anzuwenden, wenn die Variable 'noclobber' gesetzt ist und die angegebene Datei bereits existiert.

Schliesslich kann auch durch die Notation

```
Kommando >> Datei
```

ein Anhaengen des Ausgabestroms an die Datei erreicht werden. Falls in diesem Fall die Variable 'noclobber' gesetzt ist, wird eine Fehlernachricht generiert, wenn die Datei noch nicht existiert. C-Shell erzeugt dann die Datei, wenn sie noch nicht existiert. Um die Fehlermeldung bei gesetzter 'noclobber'-Variable zu unterdruecken, ist folgende Notation anzuwenden:

```
Kommando >>! Datei
```

2.6. Interne Kommandos

Das in Abschn. 2.4. beschriebene Kommando 'alias' weist neue Kommandos zu oder gibt die existierenden Definitionen aus. Ohne Argumente aufgerufen gibt es die aktuellen Alias-Definitionen aus. Es kann auch mit einem Argument angegeben werden, z.B.

```
alias ls
```

dann gibt es die aktuelle Alias-Definition fuer das Argument (in diesem Fall 'ls') aus.

Die aequivalenten Kommandos 'cd' und 'chdir' veraendern das aktuelle Arbeitsdirectory fuer C-Shell. Es ist guenstig fuer jedes Projekt, ein eigenes Directory (oder einen Directorybaum) aufzubauen, um dort alle zum Projekt gehoerenden Dateien abzulegen, z.B.:

```
%pwd
/z/bill
%mkdir newpaper
%chdir newpaper
%pwd
/z/bill/newpaper
%
```

Das Kommando 'pwd' gibt den Namen des aktuellen Directorys aus (print working directory). Dies ist gewoehnlich ein Unterdirectory des Home-Directorys. Durch Angabe von

```
chdir
```

ohne Argumente ist es moeglich zum Home-Login-Directory zurueckzukehren.

Das Kommando 'echo' gibt seine Argumente aus. Es wird haeufig in C-Shell-Skripten oder als interaktives Kommando zur Erkennung der Dateinamenexpansion benutzt.

Das Kommando 'history' gibt den Inhalt der History-Liste aus. Die in der Liste angegebenen Zahlen koennen als Verweise auf Kommandos benutzt werden, wenn es schwierig ist aus dem Kontext eine Verbindung herzustellen.

Wenn ein Ausrufzeichen (!) im Wert der Variablen 'prompt' angegeben ist, so wird es von C-Shell durch den Index in der History-Liste substituiert.

```
set prompt='\!%'
```

Diese Zahl kann als Verweis auf dieses Kommando in der History-Substitution verwendet werden. Dabei ist zu beachten, dass dem Ausrufungszeichen ein Backslash-Zeichen vorangestellt sein muss, auch innerhalb von Apostrophen.

Das Kommando 'logout' beendet ein Login-Shell, in dem die Variable 'ignoreeof' gesetzt ist.

Das Kommando 'repeat' kann zur n-maligen Wiederholung eines Kommandos benutzt werden. Um z.B. 5 Kopien der Datei 'one' in die Datei 'five' zu erzeugen, kann

```
repeat 5 cat one >> five
```

eingegeben werden.

Das Kommando 'setenv' kann benutzt werden, um Variablen in der globalen Umgebung zu setzen, z.B. wird durch

```
setenv TERM adm3a
```

der globalen Variablen TERM der Wert 'adm3a' zugewiesen. Das Kommando 'printenv' gibt alle globalen Variablen mit ihren Werten aus, z.B.

```
%printenv
HOME  /z/bill
SHELL /bin/csh
TERM  adm3a
%
```

Das Kommando 'source' kann benutzt werden, um den Inhalt einer Kommandodatei abzuarbeiten, z.B. kann durch

```
source .cshrc
```

die Ausfuehrung der Datei '.cshrc' (z.B. nach einer Aenderung) veranlasst werden. Sie wird dadurch auch wirksam, ohne das C-Shell neu gestartet werden muss.

Das Kommando 'time' veranlasst die Zeitueberwachung eines Kommandos unabhaengig davon, wieviel CPU-Zeit es belegt. Zum Beispiel zeigt

```
%time cp five five.save
0.0u 0.3s 0:01 26%
%time wc five.save
 1200 6300 37650 five.save
1.2u 0.5s 0:03 55%
%
```

dass das Kommando 'cp' weniger als ein Zehntel einer Sekunde der User-Zeit und nur drei Zehntel Sekunden der Systemzeit fuer das Kopieren benoetigte. Das Kopieren 'wc' benoetigte 1,2 Sekunden User-Zeit, 0,5 Sekunden Systemzeit bei einer Gesamtzeit von drei Sekunden fuer das Zaehlen der Zeilen, Zeichen und Woerter der Datei 'five.save'
Die Prozentangabe (55%) zeigt an, dass ueber die Zeit von drei Sekunden das Kommando 'wc' rund 55% der verfuegbaren CPU-Zyklen des Rechners ausgenutzt hat.

Die Kommandos 'unalias' und 'unset' koennen zum Streichen von Alias- und Variablendefinitionen verwendet werden.

Das Kommando 'wait' kann zum schnellen Test auf Beendigung von Hintergrundkommandos benutzt werden. Wenn C-Shell mit einem weiteren Promptzeichen reagiert, so ist die Abarbeitung von Hintergrundkommandos beendet. Andernfalls wird auf das Ende der Abarbeitung gewartet. Das Warten kann dann durch Eingabe von RUB oder DELETE abgebrochen werden. Dabei werden dann die Prozessnummern und Namen der nicht fertigen Prozesse ausgegeben.
Folgendes Beispiel soll dies verdeutlichen:

```
%nroff paper | lpr&
2450
2451
%wait
 2451 lpr
 2450 nroff
wait interrupted.
%
```

Falls es notwendig ist, einen Hintergrundprozess anzuhalten, so muss das Kommando 'kill' benutzt werden. Dabei ist die Nummer des zu stoppenden Prozesses anzugeben, z.B. wird durch

```
%kill 2450
%wait
2450: nroff: Terminated.
%
```

die Abarbeitung von 'nroff' beendet.

3. C-Shell-Steuerstrukturen und Kommandoprozeduren

3.1. Einleitung

Es ist moeglich, Kommandos in Dateien (sog. Shell-Skripten oder Kommandoprozeduren) unterzubringen und sie von dort aus einzulesen und abarbeiten zu lassen. Die zum Schreiben von C-Shell-Skripten zur Verfuegung stehenden Moeglichkeiten werden in diesem Abschnitt beschrieben.

3.2. Aufruf und Variable 'argv'

Ein C-Shell-Kommandoskript kann durch Eingabe von

```
csh script ...
```

abgearbeitet werden, wobei 'script' der Name einer Datei ist, die eine Folge von C-Shell-Befehlen enthaelt. Die drei Punkte stehen fuer Argumente. Diese Argumente werden von C-Shell der Variablen 'argv' zugewiesen, danach beginnt die Ausfuehrung. Die Argumente sind somit innerhalb der C-Shell-Prozedur wie Werte jeder anderen C-Shell-Variablen verfuegbar.

Wenn die Datei 'script' durch Eingabe von

```
chmod 755 script
```

einen ausfuehrbaren Status erhaelt, so genuegt zum Aufruf nur die Eingabe des Dateinamens.

```
script
```

Das Kommando /bin/csh wird dann automatisch aufgerufen, um die angegebene Kommandoprozedur abzuarbeiten. Wenn das erste Zeichen der ersten Zeile kein Doppelkreuz (#) ist (kennzeichnet C-Shell-Kommentar), so wird /bin/sh (Standard Bourne-Shell) aufgerufen, um die Kommandoprozedur abzuarbeiten.

3.3. Variablensubstitution

Nach Aufteilung einer jeden Eingabezeile in Woerter und Ausfuehrung der History-Substitution wird vor Ausfuehrung der verschiedenen Kommandos ein als Variablensubstitution bezeichneter Mechanismus auf die Woerter angewendet. Durch das Dollarzeichen ($) gekennzeichnete Variablennamen werden durch ihre Werte ersetzt, z.B. gibt

```
echo $argv
```

in einer Kommandoprozedur den aktuellen Wert der Variablen 'argv' ueber das Kommando 'echo' aus. Falls 'argv' keinen Wert besitzt so erfolgt eine Fehlermeldung.

Eine Reihe von Kennzeichnungen dienen dem Zugriff auf Komponenten und Attributen von Variablen. Die Notation

```
$?name
```

liefert eine 1, wenn die Variable 'name' gesetzt wurde (d.h. einen Wert besitzt) und 0, falls sie nicht gesetzt wurde (d.h. keinen Wert besitzt). Diese Notation wird demzufolge benutzt, um festzustellen ob bestimmten Variablen Werte zugewiesen wurden.
Alle anderen Formen von Verweisen auf undefinierte Variablen fuehren zu Fehlern.

Die Notation

```
$#name
```

liefert die Elementeanzahl des Wertes (d.h. die Anzahl der Zeichenkettenkomponenten). Folgendes Beispiel soll dies illustrieren:

```
%set argv=(a b c)
%echo $?argv
1
%echo $#argv
3
%unset argv
%echo $?argv
0
%echo $argv
Undefined variable: argv
%
```

Es ist ebenso moeglich, auf Komponenten des Wertes einer Variablen zuzugreifen, z.B. liefert

    $argv[1]

die erste Komponente des Wertes von 'argv', im obigen Beispiel das Zeichen 'a'.

    $argv[$#argv]

liefert die dritte Komponente, das Zeichen 'c' und

    $argv[1-2]

liefert die Zeichenkette 'ab'.

Eine andere in C-Shell-Skripten nuetzliche Notation ist

    $n

wobei n eine Integergroesse ist. Sie steht als Kurzschreibweise fuer $argv[n], den n-ten Parameter. Die Notation

    $*

ist eine Kurzschreibweise fuer $argv. Die Schreibweise

    $$

steht fuer die augenblickliche Prozessnummer. Diese Zahl ist eindeutig. Sie kann deshalb zur Erzeugung von eindeutigen temporaeren Dateinamen benutzt werden.

Ein kleiner Unterschied besteht zwischen den Schreibweisen $n und $argv[n]. Die zweite Form liefert einen Fehler, wenn n nicht im Bereich von 1 bis $#argv liegt. Die erste Form ($n) dagegen liefert diese Fehlermeldung nicht!

Es ist kein Fehler, einen Bereich der Form 'n-' anzugeben, wenn die Variable weniger als n Komponenten besitzt. Es werden dann keine Woerter substituiert. Ein Bereich der Form 'm-n' fuehrt zur Substiyution eines leeren Vektors ohne Fehlerausgaben, wenn m die Anzahl der Elemente der gegebenen Variablen uebersteigt, vorausgesetzt n liegt im Bereich.

3.4. Ausdruecke

Um C-Shell-Skripte aufbauen zu koennen, ist es notwendig, aus den Werten von Variablen Ausdruecke berechnen zu koennen. Alle aus der Sprache C bekannten Operatoren sind mit den gleichen Vorrangregeln in C-Shell verfuegbar. Die Operatoren '==' und '!=' vergleichen Zeichenketten und die Operatoren '&&' und '||' sind als

logische UND/ODER-Operatoren implementiert.
C-Shell erlaubt weiterhin durch die Notation

```
-? filename
```

den Test von Dateieigenschaften. Fuer das Fragezeichen ist je nach zu testender Eigenschaft ein entsprechendes Zeichen einzusetzen. So testet z.B. die Notation

```
-e filename
```

ob die angegebene Datei existiert. Weiterhin ist der Test des Dateistatus (read, write oder execute), der Test, ob es sich um ein Directory handelt oder der Test der Dateilaenge auf Null moeglich.

C-Shell gestattet den Test, ob ein Kommando ordnungsgemaess beendet wurde. Die Form {Kommando} liefert eine 1, wenn das Kommando normal beendet wurde (mit Exit-Status 0) und eine 0, wenn das Kommando nicht ordnungsgemaess beendet wurde (Exit_Status ungleich 0). Falls genauere Informationen ueber den Austrittsstatus eines Kommandos benoetigt werden, so ist dies ueber den Wert der Variablen 'status' moeglich. Da $status durch jedes Kommando gesetzt wird, ist der Wert der Variablen nur immer von kurzer Bedeutung. Er kann aber gerettet werden, wenn er mehrmals oder zu einem spaeteren Zeitpunkt benoetigt wird.
Eine vollstaendige Liste der verfuegbaren Ausdruecke wird unter csh(1) im WEGA-Programmierhandbuch gegeben.

3.5. Beispiel einer C-Shell-Prozedur

Die folgende Prozedur benutzt Operatoren zur Ausdrucksbildung sowie einige Ablaufstrukturen von C-Shell.

```
#
# dieser Skript copyc kopiert die als Argumente
# angegebenen C-Quellprogramme in das Directory
# ~/backup, wenn sie sich von den bereits im
# Directory ~/backup enthaltenen Dateien unterscheidet
set noglob
foreach i ($argv)
  if($i:r.c != $i) continue
  # keine .c Datei so tue nichts
  if(! -r ~/backup/$i:t) then
    echo $i:t nicht in backup
    continue
  endif
  cmp -s $i ~/backup/$i:t
  # um $status zu setzen
  if($status != 0) then
    echo neue backup von $i
    cp $i ~/backup/$i:t
  endif
end
```

Dieser Skript benutzt das Kommando 'foreach', das die zwischen dem Schluesselwort 'foreach' und dem abschliessenden 'end' befindlichen Befehle fuer jeden in den runden Klammern angegebenen Wert ausfuehrt. Der Variablen (in diesem Fall i) werden nacheinander die Werte der Liste zugeordnet. Innerhalb der Schleife ist es mit Hilfe der Anweisung 'break' moeglich die Abarbeitung abzubrechen und mit der naechsten Iteration fortzusetzen. Nach Ende der Anweisung 'foreach' behaelt die Iterationsvariable ihren letzten Wert.

Die Variable 'noglob' wurde gesetzt, um eine Dateinamenaufloesung bei Elementen von 'argv' zu verhindern. Dies ist ratsam, wenn die Argumente einer Shell-Prozedur Dateinamen sind, die bereits aufgeloest wurden oder wenn die Argumente Metazeichen zur Dateinamensaufloesung enthalten koennen. Es ist auch moeglich, jeden Gebrauch eines Metazeichens zu kennzeichnen. Dies ist aber komplizierter und weniger zuverlaessig.

Die andere in diesem Beispiel verwendete Steuerstruktur ist die if-Anweisung. Sie hat die Form

```
if (Ausdruck) then
  Kommando
  ...
endif
```

Die Anordnung der Schluesselwoerter ist bei der augenblicklichen Implementation von C-Shell nicht flexibel. Eine andere Form der if-Anweisung ist

```
if (Ausdruck) Kommando
```

sie kann auch als

```
if (Ausdruck) \
  Kommando
```

geschrieben werden. Das Newline-Zeichen ist hierbei durch das Backslash-Zeichen gekennzeichnet. Es muss direkt vor dem Zeilenende stehen. Das Kommando darf keine der folgenden Zeichen einbeziehen: |, & oder ; und auch keine weiteren Steueranweisungen.

Die mehreren if-Anweisungen im obigen Beispiel koennen durch Verwendung einer Folge von else-if Paaren gefolgt von einem 'else' und einem 'endif' auch zusammengefasst werden.

```
if (Ausdruck) then
  Kommandos
else if (Ausdruck) then
     Kommandos
...
```

```
        else
          Kommandos
        endif
```

Ein anderer wichtiger in C-Shell-Skripten benutzter Mechanismus wird durch den Doppelpunkt (:) angezeigt. Die Notation ':r' kann zur Erzeugung des Basisnamens eines Dateinamens verwendet werden. Wenn die Variable 'i' den Wert 'foo.bar' besitzt, so entfernt

```
        %echo $i $i:r
        foo.bar foo
        %
```

der Operator ':r' die Erweiterung '.bar' vom Dateinamen.

Andere Operatoren entfernen die letzte Komponente eines Pfadnamens (:h) oder streichen alles ausser der letzten Komponente (:t). Diese Operatoren werden unter csh(1) im WEGA-Programmierhandbuch vollstaendig beschrieben.
Diese Modifikationen von Zeichenketten koennen auch durch die in Abschn. 5 beschriebene Kommandosubstitution ausgefuehrt werden. Da der Gebrauch der Kommandosubstitution die Erzeugung eines neuen Prozesses bewirkt, ist die Verwendung der Doppelpunktnoation effektiver fuer diese Faelle.
Bei der aktuellen Implementation von C-Shell ist die Anzahl von Doppelpunktoperatoren auf 1 beschraenkt. So erzeugt

```
        %echo $i $i:h:t
        /a/b/c  /a/b:t
```

nicht das erwartete Ergebnis.

3.6. Weitere Ablaufstrukturen

C-Shell enthaelt die Steueranweisungen 'while' und 'switch', die aehnlich denen von C sind. Sie haben die Form

```
        while (Ausdruck)
           Kommandos
        end
```

und

```
        switch (Wort)
        case str1:
                Kommandos
                breaksw
        ...
        case strn:
                Kommandos
                breaksw
        default:
```

```
                Kommandos
                breaksw
        endsw
```

Fuer genaue Informationen siehe csh(1). C-Programmierer sollten darauf achten, dass die Anweisung 'breaksw' das Verlassen der switch-Anweisung bewirkt, waehrend 'break' die Schleifen 'while' und 'foreach' abbricht. Ein oft gemachter Fehler ist die Verwendung von 'break' anstelle von 'breaksw' in switch-Anweisungen.

C-Shell bietet auch eine goto-Anweisung zu Marken aehnlich zu C, z.B.

```
        loop:
                Kommandos
                goto loop
```

3.7. Eingabemoeglichkeiten

Von C-Shell-Prozeduren gestartete Kommandos erhalten implizit ihre Eingabe von der Standardeingabe. Dadurch ist es moeglich C-Shell-Skripte voll in Pipelines einzubinden. Fuer die interne Dateneingabe wird eine extra Notation benoetigt.
Der folgende Skript startet den Editor und streicht fuehrende Leerzeichen in den Zeilen der als Argumente angegebenen Dateien.

```
        %cat deblank
        # deblank - Streichen von fuehrenden Leerzeichen
        foreach i ($argv)
        ed - $i <<'EOF'
        1,$s/^[]*//
        w
        q
        'EOF'
        end
        %
```

Die Notation <<'EOF' zeigt an, dass die Standardeingabe fuer das Kommando 'ed' von den im C-Shell-Skript folgenden (bis zur Zeile, die 'EOF' enthaelt) Textzeilen kommt. Das Einschliessen von EOF in Apostrophe verhindert die Ausfuehrung einer Variablensubstitution fuer die Eingabezeilen. Dies wurde gewaehlt, da die Form '1,$' im Editorkommando auftaucht und verhindert werden muss, dass fuer '$' eine Variablensubstitution ausgefuehrt wird. Dies kann auch durch Kennzeichnung des Dollarzeichens durch ein Backslash-Zeichen erreicht werden, z.B.

```
        1,\$/^[]*//
```

Es ist aber zuverlaessiger die gesamten internen Daten durch Einschliessen des Begrenzungszeichens in Apostrophe

vor Variablensubstitution zu schuetzen.

3.8. Interruptbehandlung

Wenn ein C-Shell-Skript temporaere Dateien erzeugt, so ist es guenstig Interrupts einzufangen, um vor Abbruch diese Dateien streichen zu koennen. Dies kann durch die Anweisung

```
onintr label
```

erreicht werden, wobei 'label' eine Marke in der Kommandoprozedur ist. Falls ein Interrupt auftritt, so wird zur angegebenen Marke verzweigt. Dort ist es dann moeglich, temporaere Dateien zu streichen und ueber das exit-Kommando den Skript zu verlassen. Dabei kann mit einem Austrittsstatus ungleich Null, z.B.

```
exit(1)
```

der Skript verlassen werden.

3.9. Weitere Funktionen

C-Shell enthaelt weitere fuer den Schreiber von Kommandoprozeduren nuetzliche Moeglichkeiten. So koennen z.B. die Variablen 'verbose' und 'expand' oder die entsprechenden Kommandozeilenoptionen '-v' und '-x' zur Ueberwachung (trace) der Aktionen von C-Shell benutzt werden. Die Option '-n' bewirkt, dass Kommandos zwar eingelesen, aber nicht ausgefuehrt werden.

Es ist wichtig anzumerken, dass von C-Shell nur solche Kommandoprozeduren ausgefuehrt werden, die mit einem Doppelkreuz (#) beginnen (d.h. mit einem C-Shell-Kommentar). Alle anderen Kommandoprozeduren werden vom Standard-Shell (Bourne) abgearbeitet. Aehnlich uebergibt Bourne-Shell alle Skripte an C-Shell, die mit einem Doppelkreuz beginnen. Dadurch koennen Kommandoprozeduren fuer beide Kommandointerpreter (C-Shell und Shell) ohne Komplikationen miteinander koexistieren.

Ein anderer Kennzeichnungsmechanismus benutzt Anfuehrungsstriche ("). Dieser gestattet die Variablen- und Kommandosubstitution.

Make-Kommando

Man sollte nicht versuchen, Shell-Skripte fuer Aufgaben aufzubauen, die durch das Kommando 'make' realisiert werden koennen (siehe dazu make-Beschreibung). Dieses Programm kann eine Gruppe von zusammenhaengenden Dateien verwalten oder kann eine Reihe von Operationen mit diesen abhaengigen Dateien ausfuehren. Zum Beispiel koennen fuer ein grosses

Programm, das aus mehreren Dateien besteht, in einem 'makefile' die Abhaengigkeiten beschrieben werden. Dort werden die Kommandos zur Erzeugung der Programme angegeben, wenn eine Aenderung in einer Datei auftritt.
Aktivitaeten zur Ausgabe von Listingdateien, Loeschen von Directorys und Installieren von erzeugten Programmen koennen leicht in Makefiles untergebracht werden.
Ein Makefile kann auch fuer andere Anwendungen verwendet werden, so z.B. zur Verwaltung von verschiedenen Versionen eines durch nroff oder troff erzeugten Dokuments.

4. Verschiedene C-Shell-Mechanismen

4.1. Schleifen am Terminal

Die Steueranweisung 'foreach' kann auch interaktiv am Terminal zur Ausfuehrung einer Anzahl von aehnlichen Kommandos dienen. Wenn z.B. drei verschiedene Versionen von Kommandointerpretern /bin/sh, /bin/nsh und /bin/csh im System existieren und ermittelt werden soll, wieviel Personen mit welcher Shell-Version arbeiten, so kann dies durch die Kommandos

```
%grep -c nsh$ /etc/passwd
27
%grep -c csh$ /etc/passwd
34
%grep -c -v sh$ /etc/passwd
6
%
```

erreicht werden.
Eine einfache Methode, diese Angaben zu erhalten, ist die Eingabe von

```
%foreach i ('nsh$' 'csh$' '-v sh$')
? grep -c $i /etc/passwd
? end
27
34
6
%
```

C-Shell fordert durch Angabe des Fragezeichens (?) die Eingabe des Schleifenkoerpers an.

Variablen, die eine Liste von Dateinamen oder anderer Woerter enthalten, sind innerhalb von Schleifen gebraeuchlich, z.B.

```
%set a=(`ls`)
%echo $a
csh.n csh.rm
%ls
csh.n
```

```
csh.rm
%echo $#a
2
%
```

Das Kommando 'set' weist der Variablen 'a' als Wert die Liste aller Dateinamen des aktuellen Directorys zu. Danach ist eine Iteration ueber diese Namen moeglich.

Die Ausgabe eines in ` eingeschlossenen Kommandos wird von C-Shell in eine Liste von Woertern umgewandelt. Diese Kommandosubstitution wird auch innerhalb von Anfuehrungszeichen (") ausgefuehrt. Dabei wird jede nichtleere Zeile als eine Komponente der Variablen genommen, ohne dass die Zeile in Woerter, Leerzeichen und Tabs aufgeloest wird. Der Operator ':x' kann spaeter zur Aufteilung der Komponenten in Woerter dienen.

4.2. Klammern bei Argumentexpansion

Eine andere Form der Dateinamensbildung benutzt geschweifte Klammern ({ und }). Die eingeschlossenen, durch Komma getrennten Zeichenketten werden nacheinander substituiert und von links nach rechts zugewiesen, z.B. ergibt

```
A{str1,str2,str3}B
```

die Folge

```
Astr1B  Astr2B  Astr3B
```

Diese Expansion erfolgt, bevor andere Dateinamensexpansionen ausgefuehrt werden. Sie kann auch verschachtelt werden. Die Ergebnisse einer jeden expandierten Zeichenkette werden separat von links nach rechts sortiert, z.B. erzeugt

```
mkdir ~/{hdrs, retrofit, csh}
```

die drei Unterdirectorys 'hdrs', 'retrofit' und 'csh' im Nutzer-Home-Directory. Dieser Mechanismus ist guenstig, wenn der gemeinsame Teil der Dateinamen laenger ist, wie z.B. in

```
chown bin /usr/{bin/{ex,edit},lib/{ex1.1strings,how_ex}}
```

dies steht fuer

```
chown bin /usr/bin/ex
chown bin /usr/bin/edit
chown bin /usr/lib/ex.1strings
chown bin /usr/lib/how_ex
```

4.3. Kommandosubstitution

Ein in die Zeichen ` eingeschlossenes Kommando wird vor Ausfuehrung der Dateinamensexpansion durch den Ausgabestrom des Kommandos ersetzt. Somit ist es durch

```
set pwd=`pwd`
```

moeglich, den Namen des aktuellen Directorys in der Variablen pwd zu retten. Durch

```
ex `grep -l TRACE *.c`
```

wird der Editor ex fuer die Dateien aufgerufen, die mit '.c' enden und die Zeichenkettte TRACE im Namen enthalten. Die Kommandosubstitution wird auch bei einer Neuzuweisung der Eingabe durch << und innerhalb von Anfuehrungszeichen (") ausgefuehrt. Detaillierte Angaben dazu sind unter csh(1) zu finden.

S H E L L

Eine Einfuehrung in Shell

Vorwort

Shell ist sowohl eine Kommando- als auch Programmiersprache, die eine Verbindung mit dem Betriebssystem liefert. Dieses Papier beschreibt Shell an Hand von Beispielen. Daneben existieren noch eine Reihe von anderen Shell-Programmen fuer verschiedene Anwendungen. Der Anwender kann die ihm am komfortabelsten erscheinende Version auswaehlen. Meistens wird mit C-Shell gearbeitet, da C-Shell zusaetzliche erweiterte Moeglichkeiten fuer die interaktive Arbeit bietet.

Der erste Abschnitt beschreibt die grundlegenden Moeglichkeiten der Terminalarbeit. Im zweiten Abschnitt wird die Arbeit mit Shell-Prozeduren beschrieben. Das Beherrschen einer Programmiersprache ist fuer das Verstaendnis dieses Abschnitts hilfreich. Der letzte Abschnitt beschreibt weitergehende Moeglichkeiten von Shell. Hinweise der Form "pipe(2)" verweisen auf eine Sektion des WEGA-Programmierhandbuchs.

Inhaltsverzeichnis Seite

1. Grundlagen

1.1. Einleitung

Shell ist sowohl eine Kommando- als auch Programmiersprache. Sie bildet das Bindeglied zwischen Benutzer und dem Betriebssystem WEGA. Die Moeglichkeiten von SHELL umfassen einfache Ablaufsteuerungen (wie 'while' 'if. then ... else' 'case'), Parameteruebergabe, Variablen und Zeichenkettenersetzung.

Shell und Kommandos koennen in beiden Richtungen miteinander kommunizieren. Zeichenkettenparameter (typisch fuer Dateinamen oder Flags) koennen an Kommandos uebergeben werden. Durch die Kommandos wird ein Return-Kode gesetzt, dieser kann zur Ablaufsteuerung verwendet werden. Die Standardausgabe eines Kommandos kann als Shell-Eingabe genutzt werden.

Shell modifiziert die Umgebung in der die Kommandos laufen. Der Ein-/Ausgabe koennen Dateien zugewiesen werden. Es koennen Prozesse initiiert werden, die ueber einen sogenannten Pipe-Kanal miteinander kommunizieren. Kommandos werden durch das Durchsuchen von Inhaltsverzeichnissen gefunden. Kommandos koennen eine Eingabe entweder vom Terminal oder von einer Datei erhalten.

1.2. Einfache Kommandos

Einfache Kommandos bestehen aus einem oder mehreren Woertern, die durch Leerzeichen getrennt sind. Das erste Wort ist der Name des anzufuehrenden Kommandos; weitere Woerter werden als Argumente dem Kommando uebergeben,
z.B. ist

```
who
```

ein Kommando, das die Namen der eingeschalteten Nutzer ausgibt. Das Kommando

```
ls -l
```

gibt eine Liste der Namen, der im aktuellen Inhaltsverzeichnis befindlichen Daten aus. Das Argument -l teilt dem ls-Kommando mit, dass die Statusinformationen Groesse und Erstellungsdatum fuer jede Datei mit auszugeben sind.

1.3. Hintergrund-Kommandos

Um ein Kommando auszufuehren erzeugt Shell normalerweise einen neuen Prozess und wartet auf das Ende der Abarbeitung. Ein Kommando kann aber auch im Hintergrund laufen, d.h. ohne auf das Ende des Prozesses zu

warten. Zum Beispiel ruft die Kommandozeile

```
cc pgm.c&
```

den C-Compiler auf um die Datei pgm.c zu uebersetzen. Das abschliessende Zeichen '&' ist ein Operator, der Shell veranlasst, dieses Kommando im Hintergrund abzuarbeiten. Um einem solchen Prozess auf der Spur bleiben zu koennen, gibt Shell nach der Erzeugung seine Prozessnummer aus. Eine Liste aller augenblicklich aktiven Prozesse kann ueber das ps-Kommando erzeugt werden.

1.4. Ein-/Ausgabe-Neuzuweisung

Die meisten Kommandos erzeugen einen Ausgabestrom auf dem Standard-Ausgabegeraet (dem Terminal). Diese Ausgabe kann ebenso als Datei abgelegt werden, indem geschrieben wird

```
ls-l > Datei
```

Die Notation '> Datei' wird durch Shell interpretiert und ist kein Argument, das an ls uebergeben wird. Falls die angegebene Datei noch nicht existiert, so erzeugt Shell sie; andernfalls wird der alte Inhalt der Datei ueberschrieben. Der Ausgabestrom kann auch an den bisherigen Inhalt angehaengt werden. Dies erfolgt durch folgende Notation:

```
ls - l >> Datei
```

Die Standardeingabe fuer ein Kommando kann ebenfalls von einer Datei anstelle des Terminals kommen. Dies wird durch folgende Notation erreicht:

```
wc < Datei
```

Das Kommando wc liest von der Standardeingabe (in diesem Fall der Datei zugewiesen) und gibt die Anzahl der gefundenen Zeichen, Woerter und Zeilen aus. Wenn nur die Zeilenanzahl gewuenscht wird, dann ist

```
wc -l < Datei
```

einzugeben.

1.5. Pipelines und Filter

Die Standardausgabe eines Kommandos kann direkt mit der Standardeingabe eines anderen Kommandos durch Angabe des Pipe-Operators (|), wie in

```
ls - l|wc
```

verbunden werden. Die Kommandos werden auf diese Weise durch den Aufbau einer 'pipeline' verbunden. Die Wirkung ist die gleiche wie in

```
ls - l > Datei; wc < Datei
```

mit der Ausnahme, dass keine Dateien benoetigt werden. Statt dessen werden zwei parallel laufende Prozesse durch einen Pipe-Systemaufruf miteinander verbunden.

Die Synchronisation wird dadurch erreicht, indem wc angehalten wird, wenn keine Eingabe mehr da ist, und indem ls angehalten wird, wenn der Pipe-Kanal voll ist.

Ein Filter ist ein Kommando, welches einen Eingabestrom von der Standardeingabe liest, ihn auf irgend eine Weise verarbeitet und das Ergebnis an die Standardausgabe ausgibt.

Ein solcher Filter ist das Programm grep, das aus dem Eingabestrom die Zeilen heraussucht, die eine spezifische Zeichenkette enthalten. Z.B. gibt

```
ls | grep old
```

nur die Zeilen der Ausgabe von ls aus, die die Zeichenkette "old" enthalten. Ein anderer nuetzlicher Filter ist sort. Zum Beispiel gibt

```
who - | sort
```

eine alphabetisch sortierte Liste der eingeschalteten Nutzer aus.

Eine 'pipeline' kann auch aus mehr als zwei Kommandos bestehen. Zum Beispiel gibt

```
ls | grep old | wc -l
```

die Anzahl der Dateinamen im aktuellen Inhaltsverzeichnis, die die Zeichenkette "old" enthalten, aus.

1.6. Dateinamensbildung

Dateinamen sind fuer viele Kommandos Argumente, z.B. gibt

```
ls -l main.c
```

Informationen ueber die Datei main.c aus. Im Shell gibt es die Moeglichkeit durch Angabe von speziellen Symbolen eine Liste von Dateinamen zu erzeugen. Zum Beispiel generiert

```
ls -l *.c
```

als Argument fuer ls alle Dateinamen des aktuellen Inhaltsverzeichnisses, die mit 'c' enden. Das Zeichen * steht dabei fuer eine beliebige Zeichenkette, einschliesslich der leeren Zeichenkette. Folgende spezielle Symbole sind fuer eine abkuerzende Angabe von Dateinamen verwendbar:

* steht fuer eine beliebige Zeichenkette, einschliesslich der leeren Zeichenkette

? steht fuer genau ein beliebiges Zeichen

[..] steht fuer genau ein Zeichen aus den eingeschlossenen Zeichen; ein durch ein Minuszeichen getrenntes Paar von zwei Zeichen steht fuer alle Zeichen, die lexikalisch zwischen den Zeichen liegen.

Zum Beispiel steht

```
[a-z]*
```

fuer alle Dateinamen des aktuellen Inhaltsverzeichnisses, die mit den Kleinbuchstaben a bis z beginnen.

```
/usr/fred/test/?
```

steht fuer alle Dateinamen des Inhaltsverzeichnisses /usr/fred/test, die aus genau einem Zeichen bestehen. Falls kein Dateiname gefunden wird, der dem Muster entspricht, so wird das Muster uebergangen.
Dieser Mechanismus hilft Eingaben zu minimieren.
Zum Beispiel sucht

```
echo /usr/fred/*/core
```

alle Dateien core in (Subdirectories von /usr/fred und gibt die vollstaendigen Pfadnamen aus. Die Ausfuehrung dieses Kommandos kann laengere Zeit dauern, da alle Subdirectorys durchsucht werden muessen. Es gibt eine Ausnahme bei der Angabe eines Dateinamensmusters. Ein Punkt (.) zum Anfang eines Dateinamens muss explizit angegeben werden. Zum Beispiel gibt

```
echo *
```

alle Dateinamen des aktuellen Inhaltsverzeichnisses aus, die nicht mit einem Punkt beginnen.

```
echo .*
```

dagegen gibt alle Dateinamen aus, die mit einem Punkt beginnen. Dies vermeidet die Ausgabe des Dateinamens '.' (aktuelle Directory) und '..' (parent directory). Das ls-Kommando unterdrueckt implizit Informationen ueber die '.' und '..' Dateien.

1.7. Spezielle Symbole

Zeichen mit einer speziellen Bedeutung fuer Shell, wie <, >, *, ?, |, & und , werden Metazeichen genannt. Jedes Zeichen, dem ein Backslash (\) vorangestellt ist, verliert seine spezielle Bedeutung. Das Zeichen \ wird selbst nicht ausgegeben. So gibt

```
echo  \?
```

nur das Zeichen ? aus und

```
echo \\
```

gibt das einzelne Zeichen \ aus. Um lange Zeichenketten zu erlauben, die laenger als eine Zeile sind, wird die Folge \newline ignoriert.

Das \-Zeichen dient der Kennzeichnung einzelner Zeichen. Eine Zeichenkette von speziellen Symbolen kann durch Einschliessen in Hochkomma ihre besondere Bedeutung verlieren. Zum Beispiel gibt

```
echo xx'****'xx
```

die Zeichenkette

```
xx*****xx
```

Die so gekennzeichnete Zeichenkette kann Newlines enthalten; sie darf aber kein Hochkomma enthalten. Daneben existiert noch ein dritter Kennzeichnungsmechanismus, der Anfuehrungsstriche (") benutzt (siehe dazu Abschn. 3.5)

1.8. Promptzeichen

Wenn Shell von einem Terminal aus benutzt wird, dann wird vor dem Lesen eines Kommandos ein sogenanntes Promptzeichen ausgegeben. Implizit ist dies ein Dollarzeichen ($); es kann aber durch das PS1-Kommando veraendert werden. Zum Beispiel weist

```
PS1 = yesdear
```

dem Promptzeichen die Zeichenkette yesdear zu. Wenn ein newline eingegeben wird und weitere Eingaben benoetigt werden, so gibt Shell das Promptzeichen > aus. Dies kann durch falsche Verwendung der eben beschriebenen Kennzeichnung von speziellen Symbolen auftreten. Falls dies unvorhergesehen auftritt, bringt ein Interrupt Shell zurueck, um dann weitere Kommandos zu lesen. Dieses zweite Promptzeichen kann durch das PS2-Kommando veraendert werden. Zum Beispiel

```
PS2 = more
```

1.9. Shell und Login

Nach der Login-Prozedur wird Shell aufgerufen um vom Terminal zu lesen und auszufuehren. Wenn das User's-login Directory eine Datei mit dem Namen .profile enthaelt, dann wird vorausgesetzt, dass sie Kommandos enthaelt. Diese werden von Shell vor Dialogbeginn ausgefuehrt, d.h., die in .profile abgelegte Kommandofolge wird bei jeder login-Prozedur automatisch ausgefuehrt.

2. Shell-Prozeduren

2.1. Einleitung

Shell liest und fuehrt auch Kommandos aus, die in einer Datei enthalten sind. Zum Beispiel ruft

```
sh Datei [Argumente ...]
```

Shell auf, um aus der angegebenen Datei Kommandos zu entnehmen. Eine solche Datei wird als Kommandoprozedur, Shell-Prozedur oder Shell-Script bezeichnet. Mit dem Aufruf koennen Argumente mit uebergeben werden, auf die innerhalb der Datei mit Hilfe von Positionsparametern (z.B. $1) zugegriffen werden kann. Wenn z.B. die Datei wg die Kommandofolge

```
who | grep $1
```

enthaelt, dann ist

```
sh wg fred
```

aequivalent zu

```
who | grep fred
```

UNIX-Dateien haben drei unabhaengige Attribute: read, write und execute. Das Kommando chmod kann dazu benutzt werden, eine Datei als ausfuehrbar (execute) zu kennzeichnen. Zum Beispiel erhaelt durch

```
chmod +x wg
```

die Datei wg einen ausfuehrbaren Status.
Die Kommandos

```
wg fred
```

und

```
sh wg fred
```

sind aequivalent. Dies gestattet es, Shell-Prozeduren wie Programme zu benutzen. In jedem Fall wird ein neuer Prozess kreiert um das Kommando abzuarbeiten. Zusaetzlich zur Uebermittlung der Namen der Parameter ist die Anzahl der Parameter in der Prozedur ueber die Variable $# verfuegbar. Der Name der gerade auszufuehrenden Datei ist ueber $O verfuegbar.

Ein spezieller Shell-Parameter: $*, steht fuer alle Positionsparameter, ausser $O. Dies gibt die Moeglichkeit alle Positionsparameter zu uebermitteln, wie z.B. in

```
nroff -T450 -ms $*
```

2.2. Steueranweisung for

Innerhalb von Shell-Prozeduren werden haeufig Schleifen entsprechend der Argumente ($1, $2 ...) aufgebaut, um Kommandos fuer jedes Argument auszufuehren.

Ein Beispiel fuer eine solche Prozedur ist tel, die Datei /usr/lib/telnos durchsucht. Diese Datei enthaelt Zeilen der Form

```
fred mho123
bert mho789
```

Der Inhalt von tel ist:

```
for;
do grep $i/usr/lib/telnos; done
```

Das Kommando

```
tel fred
```

gibt die Zeilen von /mr/lib/telnos aus, die die Zeichenkette fred enthalten. nf

```
tel fred bert
```

gibt die Zeilen aus, die fred enthalten und danach die, die bert enthalten.

Die von Shell akzeptierte for-Schleife hat folgende allgemeine Form:

```
for Name in w1 w2 ...
do Kommandoliste
done
```

Kommandoliste ist eine Folge von einfachen Kommandos, die durch newlines oder Semikolon getrennt oder abgeschlossen werden. Reservierte Woerter, wie do und done werden nur im Anschluss an ein newline oder Semikolon akzeptiert. Name steht fuer eine Shell-Variable, der die Woerter w1, w2, .. nacheinander nach jedem Durchlauf der Kommandoliste zugewiesen werden. Falls in w1 w2 ..." weggelassen wird, so wird die Schleife fuer jeden Positionsparameter einmal durchlaufen, d.h., es wird " in $*" angenommen.

Ein anderes Beispiel fuer die Anwendung einer Schleife ist das create-Kommando, das folgenden Inhalt hat:

```
for i do >$i; done
```

Der Aufruf

```
        create alpha beta
```

gewaehrleistet, dass die beiden Dateien alpha und beta existieren und leer sind. Die Notation '>Datei' kann benutzt werden, um eine Datei zu kreieren oder eine schon existierende zu loeschen. Ein Semikolon (oder newline) vor done wird (aus syntaktischen Gruenden) benoetigt.

2.3. Steueranweisung case

Die case-Anweisung gestattet eine Mehrfachverzweigung. Zum Beispiel ist

```
        case $# in
             1) cat >> $1 ;;
             2) cat >> $2< $1;;
             *) echo 'usage: append [from] to' ;;
        esac
```

ein append-Kommando. Wenn es mit einem Argument aufgerufen wird

```
        append Datei
```

so ist $# die Zeichenkette 1 und die Standardeingabe wird an das Ende der angegebenen Datei durch das cat-Kommando angehaengt. Das Kommando

```
        append Datei1 Datei2
```

haengt den Inhalt der Datei1 an Datei2 an. Wenn beim Aufruf von append mehr als zwei Argumente angegeben werden, so erscheint eine Ausschrift, die die richtige Benutzung des append-Kommandos erlaeutert.

Die allgemeine Form der case-Anweisung ist:

```
        case Wort in
             Muster) Kommandoliste;;
             ....
        esac
```

Shell vergleicht in der angegebenen Reihenfolge das Wort mit jedem Muster. Falls eine Uebereinstimmung festgestellt wird, dann wird die entsprechende Kommandoliste abgearbeitet und die case-Anweisung wird beendet. Das Zeichen * steht fuer ein beliebiges Muster und kann deshalb fuer einen Zweig verwendet werden, der abzuarbeiten ist, wenn keine Uebereinstimmung festgestellt wird ("otherwise-Zweig").

Es wird keine Kontrolle durchgefuehrt um abzusichern, dass nur ein Muster mit dem case-Argument uebereinstimmt. Die erste gefundene Uebereinstimmung definiert die abzuarbeitende Kommandofolge. So werden im folgenden

Beispiel die Kommandos nach dem zweiten *) niemals ausgefuehrt

```
        case S# in
             *)... ;;
             *)... ;;
        esac
```

Ein anderes Beispiel fuer die Anwendung der case-Anweisung ist die Unterscheidung zwischen verschiedenen Formen von Argumenten. Das folgende Beispiel ist ein Fragment des cc-Kommandos:

```
        for i
        do case $i in
            -[ocs]) ... ;;
            -*) echo 'unknown flag $i' ;;
            *.c) /lib/c0, $i ... ;;
             *) echo 'unexpected argument $i' ;;
           esac
        done
```

Um Kommandos auch mehreren Mustern zuordnen zu koennen, gestattet die case-Anweisung bei der Angabe der Muster auch Alternativen, die durch das |-Zeichen getrennt werden. Zum Beispiel ist

```
        case $i in
           -x|-y) ...
        esac
```

aequivalent zu

```
        case $i in
           -[xy]) ...
        esac
```

Die uebliche Notation fuer spezielle Symbole kann bei der Musterangabe verwendet werden. So wird in

```
        case $i in
           \?) ...
```

auf Uebereinstimmung mit den Zeichen ? geprueft.

2.4. Interne Daten

Die Shell-Prozedur tel aus Abschn. 2.2. benutzte die Datei /urs/ lib/telnos, um die Daten fuer grep zu liefern. Eine Alternative dazu ist, die Daten als Bestandteil der Shell-Prozedur bereitzustellen, wie in

```
        for i
        do grep $i <<!
              ...
```

```
                fred mho123
                bert mho789
                ...
        !
        done
```

In diesem Beispiel nimmt Shell die Zeilen zwischen <<! und ! als Standardeingabe fuer grep. Das Zeichen nach << kann dabei auch ein anderes sein. Abgeschlossen werden die Daten durch eine Zeile, die das nach << folgende Zeichen enthaelt.

Bevor die Daten dem Kommando zur Verfuegung gestellt werden, erfolgt, falls erforderlich, eine Parametersubstitution. Dies wird durch die folgende Prozedur edg illustriert:

```
        ed $3 <<%
         g/$1/s//$2/g
         w
        %
```

Der Aufruf

```
        edg Zeichenkette1 Zeichenkette2 Datei
```

ist dann aequivalent zu folgendem Kommando:

```
        ed file <<%
         g/Zeichenkette1/s//Zeichenkette2/g
         w
        %
```

Es ersetzt jede Zeichenkette1 in der angegebenen Datei durch die Zeichenkette2. Die Substitution kann verhindert werden, wenn das Zeichen \ zur Kennzeichnung des besonderen Zeichens $ verwendet wird, wie in:

```
        ed $3 <<+
         1,\$s/$1/$2/g
         w
         +
```

Diese Version von edg ist aequivalent zur ersten Form, mit der Ausnahme, dass der Editor ed ein Fragezeichen (?) ausgibt, wenn die Zeichenkette1 nicht auftritt.

Die Substitution innerhalb der internen Daten kann durch Kennzeichnung des Endesymbols vollstaendig verhindert werden. Zum Beispiel

```
        grep $i <<\#
        ...
        #
```

Die internen Daten werden ohne Modifikation an grep

uebergeben. Falls keine Parametersubstitution bei den internen Daten benoetigt wird, so ist die letzte Form die effektivere.

2.5. Shell-Variable

Shell bietet die Moeglichkeit der Verarbeitung von Zeichenkettenvariablen. Variablennamen muessen mit einem Buchstaben beginnen und koennen aus Buchstaben, Ziffern und dem Unterstrich bestehen. Variablen koennen durch Anweisungen Werte zugewiesen werden, wie in

```
user=fred  box=m000  acct=mh0000
```

die den Variablen user, box und acct Werte zuweist. Einer Variablen kann auch durch

```
null=
```

die leere Zeichenkette zugewiesen werden. Auf den Wert einer Variablen kann zugegriffen werden, indem vor dem Namen ein $ gesetzt wird, z.B. gibt

```
echo $user
```

den Wert der Variablen user, in diesem Fall fred, aus.

Variablen sind als Abkuerzung fuer haeufig benutzte Zeichenketten nuetzlich. Zum Beispiel kopiert

```
b=/usr/fred/bin
mv pgm $b
```

die Datei pgm aus dem aktuellen Directory in das Directory /usr/fred/bin. Eine allgemeinere Schreibweise ist fuer die Parameter- (oder Variablen-) Substitution verfuegbar, wie in

```
echo   ${user}
```

Diese Notation ist aequivalent zu

```
echo   $user
```

und wird dann benutzt, wenn dem Parameternamen Buchstaben oder Ziffern folgen. Zum Beispiel weist

```
tmp=/tmp/ps
ps a >${tmp}a
```

die Ausgabe von ps der Datei /tmp/psa zu. Dagegen substituiert

```
ps a > $tmpa.
```

den Wert der Variablen tmpa.

Mit der Ausnahme von $? werden die folgenden Variablen durch Shell implizit gesetzt. $? wird nach jeder Kommandoausfuehrung gesetzt.

$? Der Austrittsstatus (return code) des zuletzt ausgefuehrten Kommandos als dezimale Zeichenkette. Die meisten Kommandos geben eine Null zurueck, wenn sie fehlerfrei abgearbeitet werden; andernfalls wird ein Return-Kode ungleich Null zurueckgegeben. Die Auswertung dieses Wertes kann durch eine if- oder while-Anweisung erfolgen.

$# Die Anzahl (dezimal) der Positionsparameter. Wurde z.B. im append-Kommando ausgenutzt, um die Anzahl der Parameter zu ermitteln.

$$ Die Prozessnummer (dezimal) der Shell. Da die Prozessnummer eindeutig zwischen allen existierenden Prozessen ist, kann diese Zeichenkette guenstig zur Generierung von eindeutigen temporaeren Dateinamen verwendet werden. Zum Beispiel:

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! Die Prozessnummer (dezimal) des zuletzt im Hintergrund gelaufenen Prozesses.

$- Die aktuellen Shell-Flags, wie -x und -v

Die folgenden Variablen haben fuer Shell eine spezielle Bedeutung. Deshalb duerfen diese Namen nicht allgemein benutzt werden.

$MAIL Beim interaktiven Arbeiten ueberprueft Shell die durch diese Variable spezifizierte Datei, bevor ein Promptzeichen ausgegeben wird. Wenn die angegebene Datei seit der letzten Ueberpruefung modifiziert wurde, dann erfolgt die Ausgabe der Nachricht 'you have mail' bevor das naechste Promptzeichen ausgegeben wird. Diese Variable wird normalerweise in der Datei .profile gesetzt. Zum Beispiel:

```
MAIL=/usr/mail/fred
```

$HOME Das implizite Argument fuer das cd-Kommando. Das aktuelle Directory ist der Ausgangspunkt fuer alle Dateinamenverweise, die nicht mit einem / beginnen. Es kann durch das cd-Kommando veraendert werden. Zum Beispiel:

```
cd/usr/fred/bin
```

Das cd-Kommando ohne Argument entspricht

Diese Variable wird in der Anwender Login-profile-Datei gesetzt.

$PATH Enthaelt eine Liste von Directorys, in denen sich Kommandos befinden (der Suchpfad). Jedesmal, wenn durch Shell ein Kommando ausgefuehrt werden soll, so werden diese Directorys nach dem ausfuehrbaren Kommando durchsucht. Falls $PATH nicht gesetzt wurde, so werden implizit das aktuelle Directory, dann /bin und/usr/bin durchsucht. Andernfalls besteht $PATH aus Directory-Namen, die durch : getrennt sind. Zum Beispiel kennzeichnet

```
PATH=:/usr/fred/bin:/bin:usr/bin
```

das aktuelle Directory (die Nullzeichenkette vor:), /usr/fred/bin, /bin und /usr/bin, d.h., die Directorys werden in der angegebenen Reihenfolge durchsucht. Nutzer koennen somit private Kommandos haben, die unabhaengig von aktuellen Directorys abgearbeitet werden koennen. Wenn der Kommandoname ein / enthaelt, dann erfolgt dieser Suchprozess nicht. Es wird nur ein Versuch gemacht das Kommando auszufuehren.

$PS1 Das erste Shell-Promptzeichen (implizit: $).

$PS2 Das Shell-Promptzeichen fuer weitere Eingaben (implizit: >).

$IFS Der Satz von Zeichen, der als Leer- oder Trennzeichen interpretiert wird. (siehe Abschn. 3.5.)

2.6. Kommando test

Das Kommando test, selbst nicht Bestandteil von Shell, wird von Shell-Prozeduren benutzt, z.B. uebergibt

```
test -f Datei
```

den Wert 0, wenn die Datei existiert, andernfalls einen Wert ungleich Null. Im allgemeinen testet das Kommando eine Bedingung und uebergibt als Ergebnis seinen Austrittsstatus. Einige haeufig benutzte Argumente von test sind:

test s	ist wahr, wenn das Argument nicht die leere Zeichenkette ist
test -f Datei	ist wahr, wenn Datei existiert
test -r Datei	ist wahr, wenn Datei lesbar ist
test -w Datei	ist wahr, wenn Datei beschreibbar ist

test -d Datei   ist wahr, wenn Datei ein Directory ist

Eine komplette Beschreibung wird unter test(1) im Programmierhandbuch gegeben.

2.7. Steueranweisung while

Die Aktionen der for-Schleife und der case-Verzweigung werden durch Daten bestimmt, die fuer Shell verfuegbar sind. Eine while- oder until-Schleife und eine if-then-else-Verzweigung sind ebenfalls moeglich. Ihre Aktionen werden durch den Austrittsstatus von Kommandos bestimmt. Eine while-Schleife hat die allgemeine Form:

```
while Kommandoliste1
do
   Kommandoliste2
done
```

Der durch das while-Kommando abgetestete Wert ist der Austrittsstatus des letzten nach while folgenden einfachen Kommandos. Bei jedem Iterationsschritt wird die Kommandoliste2 abgearbeitet. Die Schleife wird beendet, wenn ein Austrittsstatus ungleich Null uebergeben wird. Zum Beispiel ist

```
while test $1
do ...
   shift
done
```

aequivalent zu

```
for i
do ...
done
```

shift ist ein Shell-Kommando, das die Positionsparameter $2, $3, ... in $1, $2, ... umbenennt und dabei $1 uebergeht.

Eine andere Anwendung von while/until ist das Abwarten eines externen Ereignisses. In einer until-Schleife, wird die Abbruchbedingung umgedreht, z.B. wartet

```
until test -f Datei
do sleep 300; done
Kommandos
```

darauf, dass die Datei existiert. Innerhalb des Iterationsschrittes wird vor dem neuen Test fuenf Minuten gewartet.

2.8. Steueranweisung if

Als allgemeine Form der Verzweigung ist

```
        if      Kommandoliste 1
        then    Kommandoliste 2
        else    Kommandoliste 3
        fi
```

verfuegbar, die den Wert des letzten einfachen Kommandos der ersten Kommandoliste testet. Die if-Anweisung kann in Verbindung mit dem test-Kommando dazu benutzt werden, um die Existenz einer Datei zu ermitteln.

```
        if      test-f Datei
        then    Verarbeite die Datei
        else    Tue etwas anderes
        fi
```

Ein Mehrfachtest der Form:

```
        if ...
        then ...
        else if ...
             then ...
             else if ...
                  ...
                  fi
             fi
        fi
```

Kann unter Ausnutzung der erweiterten if-Notation auch als:

```
        if ...
        then ...
        elif ...
        then ...
        elif ...
        ...
        fi
```

Das folgende Beispiel ist das touch-Kommando, welches die Zeit der letzten Modifikation fuer die angegebenen Dateien aendert. Das Kommando kann in Verbindung mit make(1) dazu benutzt werden, eine angegebene Liste von Dateien neu zu uebersetzen.

```
        flag=
        for i
        do case $i in
            -c) flag=N ;;
             *) if test -f $i
                then ln $i junk$$; rm junk$$
                elif test $flag
                then echo file \'$i\' does not exist
                else >$i
```

```
                fi
            esac
        done
```

Das -c Flag in diesem Kommando veranlasst, dass die nachfolgenden Dateien kreiert werden, falls sie noch nicht existieren. Andernfalls (d.h. ohne -c) wird eine Fehlernachricht ausgegeben, wenn die Datei nicht existiert. Die Shell-Variable flag wird in irgendeiner nichtleeren Zeichenkette gesetzt, wenn das -c-Argument auftaucht. Die Kommandos

```
ln ...; rm ...
```

stellen eine Verbindung zur Datei her (link) und loeschen diese sofort wieder; dies dient zur Aktualisierung des letzten Zugriffsdatums. Die Folge

```
if     Kommando1
then   Kommando2
fi
```

kann auch als

```
Kommando1   &&  Kommando2
```

geschrieben werden.

Umgedreht wird bei

```
Kommando1   ||  Kommando2
```

das Kommando2 nur ausgefuehrt, wenn der Austrittsstatus von Kommando1 ungleich 0 ist. In jedem Fall wird das letzte einfache Kommando ausgewertet.

2.9. Kommandogruppierung

Kommandos koennen in zwei Arten gruppiert werden:

```
{ Kommandoliste; }
```

und

```
( Kommandoliste )
```

Bei der ersten Form wird die Kommandoliste einfach abgearbeitet. Die zweite Form veranlasst die Abarbeitung der Kommandofolge als separaten Prozess. Zum Beispiel fuehrt

```
( cd x; rm junk )
```

das Kommando rm junk in dem Directory x aus, ohne das aktuelle Directory zu veraendern. Die Kommandos

```
cd x; rm junk
```

haben denselben Effekt, danach ist das aktuelle Directory aber x.

2.10. Austesten von Shell-Prozeduren

Shell bietet zwei Kontrollmechanismen, um das Testen von Shell-Prozeduren zu unterstuetzen. Der erste kann durch die Anweisung

```
set -v   (v fuer verbose)
```

innerhalb der Shell-Prozedur eingeschaltet werden. Er veranlasst die Ausgabe der Zeilen der Prozedur, wenn sie gelesen werden. Das unterstuetzt das Finden von einzelnen syntaktischen Fehlern. Es kann auch ausserhalb der Prozedur durch die Angabe der folgenden Kommandozeile eingeschaltet werden

```
sh -v proc ...
```

wobei proc der Name einer Shell-Prozedur ist. Dieses Flag kann in Verbindung mit dem u-Flag, das dann die Ausfuehrung weiterer Kommandos verhindert, benutzt werden.
(Dieses Flag kann durch set -n gesetzt werden, das Terminal braucht im Abbruchfall die Eingabe eines end-of-file (EOF), um weiter arbeiten zu koennen.) Das Kommando

```
set -x
```

veranlasst die Ausgabe aller Parametersubstitutionen und eines jeden Kommandos, das ausgefuehrt wird. Beide Flags koennen durch Eingabe von

```
set -
```

ausgeschaltet werden. Der aktuelle Stand der Shell-Flags ist ueber die Variable $ verfuegbar.

2.11. Das man-Kommando

Das man-Kommando kann zur Ausgabe von Teilen eines Dokumentes benutzt werden. Es wird z.B. so aufgerufen:

```
man sh
man -t ed
man 2 fork
```

Durch die erste Zeile wird der zu sh gehoerige Teil des Online-Manuals ausgegeben. Da keine Bereichsnummer spezifiziert wurde, wird Teil 1 (Kommandos) benutzt. Das zweite Beispiel gibt einen Fotosatz (option -t) der

Editorbeschreibung (ed) unter Nutzung des Programms troff aus. Das letzte Beispiel gibt das Handbuch zu fork aus Teil 2 (Systemaufrufe) aus.

Ein Beispiel fuer das man-Kommando wird im folgenden angegeben:

```
cd /usr/man
: 'colon is the comment command'
: 'default is nroff($N), section 1 ($s)'
N=n  s=1
for i
do case $i in
   [1-9*]) s=$i ;;
   -t)     N=t;;
   -n)     N=n;;
   -*)     echo unknown flag \'si\';;
    *)     if test -f man$s/$i.$s
           then ${N}roff man0/${N}aa man$s/$i.$s
           else: 'look through all manual sections'
                 found=no
                 for j in 1 2 3 4 5 6 7 8 9
                 do if test -f man $j/$i.$j
                    then man $j $i
                         found=yes
                    fi
                 done
                 case $found in
                       no) echo '$i: manual page not found'
                 esac
           fi
   esac
done
```

3. Schluesselwortparameter

3.1. Einleitung

Shell-Variablen werden durch Zuweisungen oder den Eintritt in eine Shell-Prozedur Werte zugewiesen. Ein Argument der Form Name=Wert, das dem Prozedurnamen vorangestellt ist, bewirkt eine Wertzuweisung, bevor die Shell-Prozedur gestartet wird. Der zugewiesene Wert wird in der Shell-Prozedur nicht veraendert; z.B. fuehrt

```
user=fred command
```

die Shell-Prozedur command aus, wobei vorher die Variable user den Wert fred erhaelt. Argumente der Form Name=Wert koennen an beliebiger Stelle der Argumentliste stehen, wenn davor das Flag -K angegeben wird. Solche Namen werden Schluesselwortparameter genannt. Andere verbleibende Argumente sind weiterhin ueber die Positionsparameter $1, $2 usw. erreichbar.

Das set-Kommando kann ebenfalls benutzt werden, um Positionsparameter aus dem Inneren einer Prozedur zu setzen. Zum Beispiel steht

```
set - *
```

$1 auf den ersten Dateinamen in dem aktuellen Diectory, $2 auf den naechsten usw. Das erste Argument (-) sichert die korrekte Behandlung, wenn der erste Dateiname mit einem Minuszeichen beginnt.

3.2. Parameter-Uebermittlung

Beim Aufruf einer Shell-Prozedur koennen sowohl Positions- als auch Schluesselwortparameter bereitgestellt werden. Schluesselwortparameter sind implizit Shell bekannt, wenn sie exportiert wurden. Zum Beispiel markiert das Kommando

```
export user box
```

die Variablen user und box fuer einen Export; d.h. wenn eine Shell-Prozedur aufgerufen wird, so werden Kopien von allen exportierten Variablen fuer die Benutzung innerhalb der Prozedur angefertigt. Durchgefuehrte Modifikationen dieser Variablen durch die Prozedur aendern nicht den Wert der Variablen fuer Shell. Eine Shell-Prozedur kann nicht ohne explizite Anforderung den Status seines Aufrufers aendern. Gemeinsame Dateideskriptoren sind eine Ausnahme von dieser Regel. Namen (Variablen) deren Werte sich nicht veraendern sollen (duerfen), koennen als readonly vereinbart werden. Die Form ist dieselbe wie beim export-Kommando

```
readonly name ...
```

Nachfolgende Versuche readonly-Variablen zu veraendern sind nicht gestattet.

3.3. Parametersubstitution

Wenn einem Shell-Parameter kein Wert zugewiesen wurde, so wird die leere Zeichenkette fuer ihn substituiert. Wenn die Variable d z.B. nicht gesetzt ist, so gibt

```
echo $d
```

oder

```
echo ${d}
```

nichts aus. Eine implizite Zeichenkette kann mit angegeben werden, wie in

```
echo ${d-.}
```

Es wird der Wert der Variablen d ausgegeben, wenn sie gesetzt ist und andernfalls ".". Die implizite Zeichenkette wird unter Beruecksichtigung der Kennzeichnung der speziellen Symbole ermittelt, so dass

```
echo ${d-'*'}
```

das Zeichen * ausgibt, wenn die Variable d nicht gesetzt ist. Aehnlich kann durch

```
echo s{d-$1}
```

der Wert $1 (erste Positionsparameter) ausgegeben werden (falls vorhanden), wenn d keinen Wert besitzt. Einer Variablen kann durch folgende Notation ein impliziter Wert zugewiesen werden:

```
echo ${d=.}
```

diese Anweisung substituiert die gleiche Zeichenkette wie

```
echo ${d-.}
```

und weist d, wenn es nicht schon vorher gesetzt wurde, die Zeichenkette zu. Die Zuweisungsnotation ${...=...} ist fuer Positionsparameter nicht verwendbar.

Die Notation

```
echo ${d?Nachricht}
```

bewirkt, dass der Wert der Variablen d ausgegeben wird, falls sie einen hat, und andernfalls wird die angegebene

Nachricht ausgegeben und die Shell-Prozedur verlassen. Falls keine Nachricht spezifiziert wird, so wird eine Standardnachricht ausgegeben. Es folgt ein Beispiel fuer eine Shell-Prozedur die einige vorher gesetzte Parameter benoetigt:

```
: ${user?} ${acct?} ${bin?}
...
```

Der Doppelpunkt (:) ist ein in Shell eingebautes Kommando, das nichts weiter leistet, als dass es die Argumentwerte ermittelt. Falls eine der Variablen user, acct oder bin nicht gesetzt ist, wird die Abarbeitung der Shell-Pozedur abgebrochen.

3.4. Kommandosubstitution

Die Standardausgabe eines Kommandos kann in einer der Parametersubstitution aehnlichen Weise substituiert werden. Das Kommando purt gibt ueber seine Standardausgabe den Namen des aktuellen Directorys aus. Wenn z.B. das aktuelle Directory /usr/fred/bin ist, so ist die Zuweisung

```
d=`pwd`
```

aequivalent zu

```
d=/usr/fred/bin
```

Die gesamte Zeichenkette zwischen ` und ` wird als Kommando genommen, das ausgefuehrt wird und dann durch die Ausgabe ersetzt wird. Das Kommando wird unter Ausnutzung der normalen Notation fuer die speziellen Symbole geschrieben, mit der Ausnahme, dass einem ` ein \ vorangestellt sein muss; z.B. ist

```
ls `echo "$1"`
```

aequivalent zu

```
ls $1
```

Die Kommandosubstitution wird in allen Zusammenhaengen ausgefuehrt, in denen Parametersubstitution, einschliesslich interner Daten, auftritt. Die Behandlung des Ergebnistextes ist in beiden Faellen die gleiche. Dieser Mechanismus gestattet es, zeichenkettenverarbeitende Kommandos innerhalb von Shell-Prozeduren zu benutzen. Ein Beispiel fuer ein solches Kommando ist basename, das spezielle Endungen von Zeichenketten entfernt; z.B. gibt

```
basename main.c
```

die Zeichenkette main aus. Die Ausnutzung wird im folgenden Fragment des cc-Kommandos illustriert.

```
        case $A in
        ...
        *.c)   B=`basename $A .c`
        ...
        esac
```

Hierbei wird B auf den Teil von $A ohne die Endung .c gesetzt. Es folgen einige weitere Beispiele:

```
        for i in `ls -t`; do ...
        Die Variable i wird auf die Dateinamen
        (in der zeitlichen Reihenfolge)
        des aktuellen Directorys gesetzt.

        set `date`; echo $6 $2 $3, $4
        gibt z.B. aus    1981 Nov 1, 23:59:59
```

3.5. Berechnung und Kennzeichnung spezieller Symbole

Shell ist ein Makroprozessor mit den Moeglichkeiten der Parametersubstitution, der Kommandosubstitution und der Dateinamensgenerierung fuer Kommandoargumente. Dieser Abschnitt erlaeutert die Berechnungsreihenfolge und die Effekte der verschiedenen Kennzeichnungsmoeglichkeiten.

Kommandos werden entsprechend der Grammatik aufgegliedert. Vor Ausfuehrung eines Kommandos werden folgende Substitutionen vorgenommen:

- Parametersubstitution: z.B. $user

- Kommandosubstitution: z.B. `pwd`
  Es wird nur eine Berechnung ausgefuehrt, wenn z.B. der Wert der Variablen x die Zeichenkette "$y" ist, dann gibt
  ```
        echo $x
  ```
  $y aus.

- Leerzeicheninterpretation:
  Nach den oben angefuehrten Substitutionen werden die Zeichen in Woerter zerlegt, die keine Trennzeichen enthalten (blank interpretation). Trennzeichen sind Zeichen, die in der Zeichenkette $IFS angegeben sind. Implizit Leerzeichen, Tabulator und Zeilenvorschub. Die leere Zeichenkette wird nicht als Wert betrachtet, ausser das sie gekennzeichnet ist; z.B. uebergibt
  ```
        echo ''
  ```
  als erstes Argument die leere Zeichenkette an echo, dagegen ruft die Zeile
  ```
        echo $null
  ```
  echo ohne Argument auf, wenn die Variable null nicht gesetzt ist oder die leere Zeichenkette enthaelt.

- Dateinamenerzeugung:
  Jedes Wort wird nach den speziellen Mustersymbolen

(Komma), Fragenzeichen und [...] durchsucht, und es wird eine alphabetische Liste von Dateinamen erzeugt, die das Wort ersetzt. Ein jeder von diesen Dateinamen ist ein extra Argument.

Die eben beschriebene Rechnung wird ebenfalls in der Liste von Woertern, die mit einer for-Schleife verbunden sind, angewendet. Eine Substitution erfolgt in dem fuer eine case-Verzweigung verwendeten Wort.

Zusaetzlich zu dem eben beschriebenen Kennzeichnungsmechanismus gibt es eine dritte Kennzeichnungsart. Diese verwendet Anfuehrungsstriche. Innerhalb von Anfuehrungszeichen (") wird die Parameter- und Kommandosubstitution durchgefuehrt, dagegen erfolgt keine Dateinamenserzeugung und Leerzeicheninterpretation. Die folgenden Zeichen haben innerhalb von Anfuehrungszeichen eine besondere Bedeutung, die sie durch Vorsetzen eines \ verlieren.

$	Parametersubstitution
`	Kommandosubstitution
"	beendet die besondere Zeichenkette
\	wird zur Kennzeichnung der besonderen Zeichen $ ` " und \ benutzt.

z.B. uebergibt

```
echo "$x"
```

den Wert der Variablen x als einzelnes Argument an echo. Aehnlich uebermittelt

```
echo "$*"
```

die Positionsparameter als einzelnes Argument und ist aequivalent zu

```
echo "$1 $2 ..."
```

Die Notation $@ ist die gleiche wie $*, mit der Ausnahme bei der besonderen Kennzeichnung. Das Kommando

```
echo "$@"
```

uebermittelt die Positionsparameter unberechnet an echo und ist aequivalent zu

```
echo "$1" "$2" ...
```

Die folgende Tabelle gibt fuer jeden Kennzeichnungsmechanismus die Shell-Metazeichen an, die ermittelt werden.

	Metazeichen					
	\	$	*	`	"	'
'	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	y	t

t Terminator
y wird interpretiert
n nicht interpretiert

In den Faellen, wo mehr als eine Berechnung fuer eine Zeichenkette benoetigt wird, wird das eingebaute eval-Kommando benutzt. Wenn z.B. die Variable x den Wert $y hat und y den Wert pgr besitzt, dann gibt

```
eval echo $x
```

die Zeichenkette pgr aus.

Das eval-Kommando berechnet seine Argumente (wie alle Kommandos) und behandelt das Ergebnis als Eingabe fuer Shell. Die Eingabe wird gelesen und das (die) Ergebniskommando(s) ausgefuehrt. Zum Beispiel ist

```
wg = 'eval who|grep'
$wg fred
```

aequivalent zu

```
who|grep fred
```

In diesem Fall wird eval benoetigt, da nach der Substitution die Metazeichen (wie |) nicht interpretiert werden.

3.6. Fehlerbehandlung

Die Art der Fehlerbehandlung haengt von der Fehlerart und davon ab, ob Shell interaktiv benutzt wird oder nicht. Shell wird interaktiv benutzt, wenn sein Ein- und Ausgabestrom mit dem Terminal verbunden sind (wie durch gtty(2) bestimmt). Mit dem i-Flag aufgerufen, ist Shell ebenfalls interaktiv.

Die Ausfuehrung eines Kommandos (Abschn. 3.8) kann aus einem der folgenden Gruende misslingen:

- Ein-/Ausgabe-Neuzuweisungen misslingen, da z.B. Dateien nicht existieren oder sie nicht erzeugt werden koennen.

- Das Kommando existiert nicht oder kann nicht ausgefuehrt werden.

- Das Kommando wird nicht ordnungsgemaess beendet; z.B. durch einen "bus error" oder "memory fault" (siehe Bild 3-1 fuer eine komplette Liste der wega-Signale)

- Das Kommando wird normal beendet, liefert aber einen Rueckgabewert (Status) ungleich Null.

In all diesen Faellen fuehrt Shell mit der Ausfuehrung des naechsten Kommandos fort. Mit Ausnahme des letzten Falles wird durch Shell eine Fehlernachricht ausgegeben. Alle restlichen Fehler fuehren zu einem Abbruch der Kommandoprozedur. Eine interaktive Shell kehrt dann zurueck, um ein weiteres Kommando vom Terminal zu lesen. Solche Fehler sind:

- Syntaxfehler, z.B. if...then...done

- Ein Signal wie z.B. Interrupt; Shell wartet auf die Ausfuehrung des aktuellen Kommandos, wenn eins da ist, und beendet entweder seine Arbeit oder kehrt zum Terminaldialog zurueck.

- Fehlerhafte Verwendung eines eingebauten Kommandos, wie z.B. cd.

Das Shell-Flag -e benutzt den Abbruch, sobald ein beliebiger Fehler erkannt wird.

Bild 3-1 wega-Signale:

Die mit einem Stern markierten Signale veranlassen einen Speicherabzug (core dump), wenn sie nicht eingefangen werden. Shell jedoch selber ignoriert quit, welches das einzige externe Signal ist, das einen Dump benutzt. Die fuer Shell potentiell interessanten Signale der Liste sind 1, 2, 3, 14 und 15.

```
 1      hangup
 2      interrupt
 3*     quit
 4      illegal instruction
 5*     trace trap
 6*     IOT system call
 7*     Unused (frueher EMT instruction)
 8*     floating point exception
 9      Kill (cannos be ranght or, ignored)
10*     Unused (frueher bus error)
11*     Segmentation violation
12*     bad argument to system call
13      write on pipe with no one to read it
14      alarm clock
15      software termination (from Kill (1))
16      unassigned
```

3.7. Traps

Shell-Prozeduren werden normalerweise beendet, wenn ein Interrupt vom Terminal aktiviert wird. Das trap-Kommando wird benutzt, wenn bestimmte Aktionen ausgefuehrt werden sollen, wie z.B. das Streichen von temporaeren Dateien. So setzt

```
trap 'rm /tmp/ps$$; exit' 2
```

einen Trap fuer das Signal 2 (Terminalinterrupt). Wenn dieses Signal aktiviert wird, dann werden die Kommandos

```
rm /tmp/ps$$; exit
```

ausgefuehrt.

exit ist ein weiteres eingebautes Kommando, das die Ausfuehrung einer Shell-Prozedur beendet. Das exit-Kommando wird benoetigt, andernfalls setzt Shell nach der Trap-Behandlung an der Stelle fort, wo es unterbrochen wurde.

WEGA-Signale koennen auf drei verschiedene Weisen behandelt werden. Sie koennen ignoriert werden, in diesem Fall wird das Signal nicht an den Prozess uebermittelt. Sie koennen eingefangen werden. In diesem Fall muss der Prozess entscheiden, welche Aktion ausgefuehrt wird, wenn das Signal aktiviert wird. Sie koennen aber auch die Beendigung eines Prozesses bewirken, ohne dass weitere Aktionen ausgefuehrt werden. Wenn ein Signal mit dem Eintritt in die Shell-Prozedur ignoriert wird, z.B. durch Abarbeitung im Hintergrund (Abschn. 3.8.), so werden trap-Kommandos und das Signal ignoriert. Die Benutzung von trap wird durch eine modifizierte Version des touch-Kommandos im Bild 3-2 illustriert. Die Aktion beim Auftreten von Signalen dient zum Streichen der Datei junk$$. Das trap-Kommando erscheint vor der Erzeugung der temporaeren Datei; andernfalls koennte es fuer den Prozess moeglich sein, zu Enden, ohne die Datei zu streichen.

Da im WEGA kein Signal 0 existiert, wird es von Shell zur Anzeige der Kommandoausfuehrung beim Austritt aus der Shell-Prozedur benutzt.

Eine Prozedur selber kann Signale ignorieren, indem die leere Zeichenkette als Argument fue trap verwendet wird. Das folgende Fragment ist aus dem nohup-Kommando ausgewaehlt:

```
trap '' 1 2 3 15
```

Es ignoriert innerhalb der Prozedur und bei aufgerufenen Kommandos die Signale hangup, interrupt, quit und Kill.

Bild 3.2. touch-Kommando:

```
        flag=
        trap 'rm -f junk$$; exit' 1 2 3 15
        for;
        do case $i in
        -c)     flag=N;;
        *)      if test - f $i
                then ln $i junk$$; rm junk$$
                elif test $flag
                then echo file \'$i\' does not exist
                else >$i
            esac
        done
```

Bild 3.3. scan-Kommando:

```
        d=`pwd`
        for i in *
        do if test -d $d/$i
        then cd $d/$i
                while echo "$i:"
                        trap exit 2
                        read x
                do trap : 2; eval $x; done
           fi
        done
```

Traps koennen durch Eingabe rueckgesetzt werden. Zum Beispiel setzt Traps fuer die Signale 2 und 3 auf ihre impliziten Werte zurueck. Eine Liste der aktuellen Trapwerte kann durch Eingabe von

```
        trap
```

ermittelt werden. Die Prozedur scan (Bild 3.3) ist ein Beispiel fuer die Anwendung eines trap-Kommandos, ohne dass exit verwendet wird. scan nimmt jedes Directory in dem aktuellen Directory, gibt den Namen aus und fuehrt dann eingegebene Kommandos aus, solange bis ein End-of-file (EOF) oder ein Interrupt erscheint. Interrupts werden bei Kommandoabarbeitung ignoriert, veranlassen aber einen Abbruch, wenn scan auf Eingaben wartet.

read x ist ein eingebautes Kommando, es liest eine Zeile von der Standardeingabe und weist sie der Variablen x zu. Es wird ein Returnstatus ungleich Null uebergeben, wenn End-of-File gelesen wird oder ein Interrupt erscheint.

3.8. Kommandoausfuehrung

Um andere als eingebaute Kommandos auszufuehren, kreiert Shell zuerst einen neuen Prozess mit Hilfe des Systemaufrufs fork. Die Ausfuehrumgebung fuer das Kommando umfasst Eingabe, Ausgabe und die Zustaende der Signale und wird im Tochterprozess errichtet, bevor das Kommando ausgefuehrt wird. Das eingebaute Kommando exec wird in den seltenen Faellen benutzt, wo kein fork benoetigt wird und ersetzt Shell durch das neue Kommando. Eine einfache einfache Version des nohup-Kommandos kann so aussehen:

```
trap '' 1 2 3 15
exec $*
```

Die trap-Anweisung schaltet alle angegebenen Signale aus, so dass sie durch alle nachfolgend erzeugten Kommandos ignoriert werden. Die exec-Anweisung ersetzt Shell durch das spezifische Kommando. Im folgenden steht word nur fuer Parameter und Kommandosubstitutionen. Es wird keine Dateinamenerzeugung oder Trennzeicheninterpretation durchgefuehrt, so dass z.B.

```
echo... > *.c
```

seine Ausgabe in einer Datei mit dem Namen .c ablegt. Ein-/Ausgabezuweisungen werden von links nach rechts, so wie sie im Kommando auftreten, berechnet.

>word Die Standardausgabe (Dateideskriptor 1) wird in die Datei word geschrieben, die erzeugt wird, wenn sie noch nicht existiert.

>>word Die Standardausgabe wird in die Datei word geschrieben. Die Ausgabe wird an den bisherigen Inhalt angehaengt. Falls die Datei noch nicht existiert, so wird sie erzeugt.

<word Die Standardeingabe (Dateideskriptor 0) wird von der Datei word genommen.

<<word Die Standardeingabe wird von den folgenden Shell-Eingabezeilen genommen, bis zu (aber nicht einschliesslich) einer Zeile, die nur aus word besteht. Falls word speziell gekennzeichnet ist (quoted), so erfolgt keine Interpretation des Inhalts. Falls word nicht gekennzeichnet ist, so erfolgt eine Parameter- und Kommandosubstitution und das Zeichen \ wird zur Kennzeichnung der Zeichen \, $ und `, und des ersten Zeichens von word benutzt. Im letzten Fall wird \newline ignoriert.

>&digit Der Dateideskriptor digit wird unter Ausnutzung des Systemaufrufs dup (2) dupliziert, und das Ergebnis wird als Standardausgabe benutzt.

<&digit Die Standardeingabe wird dupliziert vom Dateideskriptor digit.

<&- Die Standardeingabe wird geschlossen.

>&- Die Standardausgabe wird geschlossen.

Den oben anegebenen Konstruktionen kann eine Zahl vorangestellt sein, um einen speziellen Dateideskriptor, anstelle der impliziten Werte 0 und 1, zu erzeugen. Zum Beispiel wird durch

```
...2> Datei
```

die Ausgabe der angegebenen Datei mit dem Dateideskriptor 2 zugewiesen. Durch

```
...2>&1
```

werden die Standardausgabe und die Kommandoausgabe verschmolzen. Der Dateideskriptor 2 wird durch Duplizierung des Dateideskriptors 1 erzeugt; der Effekt ist gewoehnlich die Verschmelzung der beiden Stroeme.

Die Umgebung fuer ein im Hintergrund laufendes Programm, wie

```
list *.c | lpr&
```

kann auf zwei Wegen modifiziert werden. Zuerst, dass die Standardeingabe fuer ein solches Programm die leere Datei /dev/null ist. Dies verhindert, dass zwei parallel laufende Prozesse (Shell und das Kommando) versuchen, von der gleichen Eingabe zu lesen. Zum Beispiel erlaubt

```
ed file &
```

beiden, dem Editor und Shell, zur gleichen Zeit von der gleichen Eingabe zu lesen. Die andere Moeglichkeit, die Umgebung fuer ein im Hintergrund laufendes Kommando zu veraendern, ist das Ausschalten der QUIT- und INTERRUPT-Signale, so dass sie durch das Kommando ignoriert werden. Dies erlaubt es, diese Signale zu verwenden, ohne dass dadurch der Hintergrundprozess beendet wird. Aus diesem Grund gilt fuer ein Signal die Vereinbarung, dass, wenn es auf 1 gesetzt wurde (d.h. ignorieren), es nicht mehr zu veraendern ist. Das Shell-Kommando trap hat keine Bedeutung fuer ein ignoriertes Signal.

3.9. Aufruf von Shell

Die folgenden Flags werden durch Shell beim Aufruf interpretiert. Wenn das erste Zeichen des nullten Arguments ein Minuszeichen ist, dann werden die in der Datei .profile abgelegten Kommandos ausgefuehrt.

-c Zeichenkette
: Wenn das c-Flag vorhanden ist, so werden in der Zeichenkette die angegebenen Kommandos ausgefuehrt.

-s
: Wenn das s-Flag vorhanden ist, oder keine Argumente angegeben wurden, so werden Kommandos von der Standardeingabe gelesen. Die Shell-Ausgaben werden an den Dateideskriptor 2 ausgegeben.

-i
: Wenn das i-Flag vorhanden ist, oder die Shell-Ein-/Ausgabe mit dem Terminal verbunden ist (durch gtty), dann ist Shell interaktiv. In diesem Fall wird das Signal TERMINATE ignoriert, so dass kill 0 eine interaktive Shell nicht abbricht. Interrupts werden eingefangen und solange ignoriert, bis wait unterbrechbar ist. In allen Faellen wird das Signal QUIT durch Shell ignoriert.

E X / V I

Kurzbeschreibung der Editoren EX und VI

Inhaltsverzeichnis Seite

1. Allgemeine Hinweise zu den Editoren unter WEGA

Es existieren vier Editorvarianten unter WEGA: ED, EDIT und EX sind zeilenorientierte Texteditoren und VI ist ein kursororientierter Bildschirmeditor. Die Texteditoren werden zur Erstellung und Modifizierung von Textdateien unter WEGA benutzt.
ED ist der Standard-Editor. Er wird hier nicht weiter beschrieben.
EDIT ist eine Variante von EX, die fuer den Lernenden besser geeignet ist. Der Editor EX bietet gegenueber dem EDIT zusaetzliche erweiterte Moeglichkeiten (Kommandos und Optionen). Diese werden hier nicht beschrieben.
Die folgende Kommandozusammenfassung in Abschn. 2.1. enthaelt die grundlegenden Kommandos und gilt sowohl fuer EDIT als auch fuer EX.

Ausfuehrliche Beschreibungen zur umfassenden Handhabung und zum Aufruf aller Editoren sind der Online-Dokumentation (man-Kommando) zu entnehmen.

2. EDIT/EX-Kurzbeschreibung

Die Editoren arbeiten mit einem temporaeren Pufferspeicher, in dem alle durchgefuehrten Aenderungen abgelegt werden. Dieser Pufferinhalt muss vor Verlassen des Editors in die entsprechende Datei zurueckgeschrieben werden, um die alte Textversion zu aktualisieren.

Waehrend des Editiervorgangs existieren zwei Arbeitsweisen: der Kommandomodus und der Eingabemodus. Im Kommandomodus gibt der Editor EX einen Doppelpunkt als Promptzeichen aus, um anzuzeigen, dass er eine Kommandoeingabe erwartet. Der Texteingabemodus wird durch die Kommandos a (append), i (insert) und c (change) erreicht. Alle danach eingegebenen Textzeilen werden in den Puffer uebernommen. Abgeschlossen wird dieser Modus durch Eingabe einer Zeile, die nur aus einem Punkt besteht.

Die Editoren arbeiten zeilenorientiert. Den meisten Kommandos koennen eine oder zwei Zeilennummern (dann durch ein Komma getrennt) vorangestellt werden, um den Bereich zu spezifizieren, auf den sie wirken. Falls keine Zeilennummern angegeben werden, so wird immer auf die aktuelle Zeile bezug genommen.
In der Kommandozusammenfassung werden die optional moeglichen Zeilennummern durch folgende Notation angezeigt:

```
(.,.)     fuer zwei Zeilennummern
(.)       fuer eine Zeilennummer
```

Anstelle einer Zeilennummer kann auch ein einzelner Punkt (er steht fuer die aktuelle Zeile), das Dollarzeichen (es steht fuer die letzte Zeile) oder eine relative Zeilennummer (z.B.: .+3 oder .-5) angegeben werden.

2.1. Kommandos

(.)a append:
Einfuegen von Text nach der angegebenen (oder der aktuellen) Zeile. Uebergang in Texteingabemodus.

(.,.)c change:
Austauschen der angegebenen Zeilen durch die nachfolgend einzugebenden Zeilen.

(.,.)d delete:
Loeschen der angegebenen Zeilen (oder der aktuellen Zeile).

e Datei
e! Datei edit:
Editieren einer neuen Datei. Loeschen des Pufferinhalts. Es wird eine Warnmeldung ausgegeben, falls der veraenderte Inhalt noch nicht gerettet wurde. Die Form e! unterdrueckt diese Warnmeldung.

f Datei file:
Benennt die aktuelle Datei in den angegebenen Dateinamen um. Falls kein Dateiname spezifiziert wird, so erfolgt die Ausgabe des aktuellen Dateinamens.

(.,.)g/Muster/Kommandos
global:
Dursucht den angegebenen Bereich (oder den gesamten Pufferbereich) und fuehrt die spezifizierten Kommandos mit jeder Zeile, die das angegebene Muster enthaelt, durch.
Wenn fuer g
g! oder
v steht, so werden die Kommandos mit den Zeilen ausgefuehrt, die das angegebene Muster nicht enthalten

(.)i insert:
Einfuegen von Text vor der angegebenen (oder der aktuellen) Zeile. Uebergang in den Texteingabemodus.

(.,.)j join:
Verbindet die angegebenen Zeilen zu einer Zeile. Die entstehende Zeile wird ausgegeben.

(.,.)l list:
Ausgabe der angegebenen Zeilen. Dabei werden das Ende der Zeile durch "$" und Tabulatorzeichen durch "^I" gekennzeichnet.

(.,.)m Adr
move:
Verschieben der angegbenen Zeilen hinter die

durch Adr gekennzeichnete Zeile.

(.,.)nu oder
(.,.)#
number:
Ausgabe von Zeilen mit ihrer Zeilennummer.

(.)o open:
Uebergang in den VI-Editor (kursorgesteuerter Bildschirmeditor, weitere Erlaeuterungen dazu siehe Beschreibung VI-Editor). Rueckkehr in den normalen Editor-Kommandomodus durch Eingabe von 'Q'.

pre preserve:
Sichert eine Kopie des augenblicklichen Pufferinhalts im Directory /usr/preserve (siehe Option -r beim Aufruf). Diese Kommando ist im Notfall dann aufzurufen, wenn ein write-Kommando nicht richtig ausgefuehrt wird.

(.,.)p print:
Ausgabe der angegebenen Zeilen.

q quit:
Verlassen des Editors. Es wird eine Warnmeldung ausgegeben, falls der veraenderte Inhalt noch nicht gerettet wurde. In diesem Fall ist vorher das write-Kommando auszufuehren.
q! Die Form q! unterdrueckt die Warnmeldung und kehrt sofort zurueck.

(.)r Datei
read:
Liest die angegebene Datei nach der spezifizierten (oder der aktuellen) Zeile ein. Um eine Datei zum Beginn des Editorpuffers einzulesen muss 0r angegeben werden. Der Pufferinhalt wird dabei nicht zerstoert.

rec Datei
recover:
Wiederherstellen einer Kopie des Editorpuffers nach einem Systemabsturz, einem Editorabsturz oder nach Ausfuehrung eines preserve-Kommandos.

(.,.)s/str1/str2/
(.,.)s/str1/str2/gc
substitute:
Ersetzen der ersten Zeichenkette (str1) durch die zweite Zeichenkette (str2) im angegebenen Zeilenbereich (oder in der aktuellen Zeile). Wird am Ende ein g angegeben, so erfolgt der Austausch fuer jedes Auftreten von str1 in einer Zeile (global). Die Angabe von c am Ende gestattet eine Abfrage vor Ausfuehrung einer jeden Aenderung.

u
undo:
Rueckgaengigmachen der vom letzten Editorkommando durchgefuehrten Aenderungen.

(.,.)w Datei
(.,.)w! Datei
write:
Schreiben des Editorpufferinhalts in die angegebene Datei. Falls kein Dateiname spezifiziert wird, so wird in die aktuelle Datei zurueckgeschrieben. Mit der Form w! kann auch eine existierende Datei ueberschrieben werden.

(.)z Anzahl
count:
Ausgabe eines Bildschirminhalts (oder der angegebenen Zeilenanzahl) ab der spezifizierten Zeile.

!Kommando
Ausfuehren eines Kommandos vom Editor aus, ohne ihn zu verlassen.

CTRL-d
Ausgabe des naechsten Fensters des Textes (normalerweise eine halbe Bildschirmseite).

(.)<cr>
Eine Zeilennummer gefolgt von CR veranlasst die Ausgabe dieser Zeile. Ein CR alleine gibt die nach der aktuellen Zeile folgende Zeile aus.

/Muster/
Sucht im Text die naechste Zeile, die das angegebene Muster (Zeichenkette) enthaelt und gibt diese aus.

//
Wiederholt die letzte Suche.

?Muster?
Sucht im Text die vorherige Zeile, die das angegebene Muster (Zeichenkette) enthaelt.

??
Wiederholt die letzte Suche, aber in umgekehrter Richtung.

3. VI-Kurzbeschreibung

Der VI-Editor kann vom EX-Editor ueber das open-Kommando (o) oder durch Eingabe von vi erreicht werden. Er kann aber auch direkt

```
%vi Dateiname(n)
```

aufgerufen werden. Dabei koennen dem Dateinamen noch verschiedene Optionen vorangestellt werden. Eine umfassende Beschreibung des VI-Editors ist der Online-Dokumentation (man-Kommando) und der WEGA-Mikrofichedokumentation zu entnehmen.
Vom VI-Editor aus lassen sich alle Kommandos des EX-Editors ausfuehren. Dazu ist als erstes Zeichen (vor dem EX-Kommando) ein Doppelpunkt einzugeben.
Die folgende Kommandouebersicht enthaelt eine Kurzzusammenfassung der VI-Befehle.

3.1. Kommandos

- Dateimanipulation:

```
:w          Zurueckschreiben der Aenderungen
:wq         Zurueckschreiben und Verlassen des Editors
:q          Verlassen des Editors
:q!         Verlassen, ohne die vorgenommenen Aenderungen
            abzuspeichern
:e Datei    Editieren der angegebenen Datei
:e!         Reeditieren; Datei wird in den Zustand versetzt,
            den sie nach dem letzten Schreiben hatte
:e#         Editieren der Datei, die beim Aufruf vor der
            aktuellen Datei angegeben wurde (auch CTRL-^)
:w Datei    Zurueckschreiben in angegebene Datei
:w! Datei Zurueckschreiben in angegebene Datei, wenn diese
            bereits existiert (Ueberschreiben)
:!Kommando    Abarbeiten des angegebenen Kommandos
:n          Editieren der naechsten in der Kommandoliste
            angegebenen Datei
:f          Anzeige der aktuellen Datei und Zeilennummer
            (auch CTRL-g)
:sh         Aufruf von Shell (Eingabe von CTRL-d fuer
            Rueckkehr)
```

- Kursorpositionierung innerhalb der Datei:

```
CTRL-f      Fenster vorwaerts
CTRL-b      Fenster rueckwaerts
CRTL-d      halbe Fenster runter
CRTL-u      halbe Fenster hoch
[n]G        Kursor auf Zeile n (impl. letzte Zeile)
/string     naechste Zeile, die Zeichenkette enthaelt
?string     analog /string aber rueckwaerts im Text
n           Wiederholen des letzten /- oder ?-Kommandos
N           analog n, aber in entgegengesetzter Richtung
/string/+n     n-te Zeile nach Zeichenkette
```

```
?string?-n     n-te Zeile vor Zeichenkette
]]          naechste Sektion/Funktion (Zeile, die mit {
            beginnt)
[[          auf vorherige Sektionsgrenze
%           finde Klammer
```

- Markieren und Rueckkehr:

```
``          Rueckkehr zur vorherigen Position im Text
''          Kursor auf erstes "non-white-Zeichen" der Zeile
            der vorherigen Position
mx          Markieren der Position mit Buchstaben x
            (beliebiger Buchstabe des Alphabets)
`x          Kursor auf mit x markierte Position
'x          auf erstes "non-white-Zeichen" der Zeile, die
            Markierung enthaelt
```

- Zeilenpositionierung:

```
H           oberste Bildschirmzeile
M           auf Mitte des Bildschirms
L           letzte Bildschirmzeile
+           erste "non-white-Position" der naechsten Zeile
-           erste "non-white-Position" der vorherigen Zeile
RETURN      wie CR; Kursor auf Beginn naechste Zeile
j           naechste Zeile, gleiche Spalte
k           vorherige Zeile, gleiche Spalte
```

- Kursorpositionierung innerhalb einer Zeile:

```
^           erste "non-white-Position"
0           Zeilenbeginn
$           Zeilenende
l oder ->   Kursor ein Zeichen nach rechts
h oder <-   Kursor ein Zeichen nach links
CTRL-h      wie <-
space       wie ->
fx          Finde x vorwaerts
Fx          Finde x rueckwaerts
tx          auf Zeichen vor x (vorwaerts)
Tx          auf Zeichen nach x (rueckwaerts)
;           Wiederhole letzte f-,F-,t- oder T-Kommando
,           Invers von ;
[n]|        auf angegebene Spalte
```

- Woerter, Folgen, Paragraphen:

```
w           Wort vorwaerts   )
b           Wort rueckwaerts )  stoppen bei Punktuation
e           ans Wortende     )
)           an den Beginn der naechsten Folge
}           an den Beginn des naechsten Paragraphen
(           an den Beginn der vorherigen Folge
{           an den Beginn des vorherigen Paragraphen
```

```
W  )
B  )      analog w-, b-, e-Kommando ohne Stop bei
E  )      Punktuation
```

- Korrektur waehrend der Eingabe:

```
CRTl-H     Loeschen letzte Zeichen
CRTL-W     Loeschen letzte Wort
erase      Erase-Funktion, wie CRTL-H
kill       Kill-Funktion, Loeschen Eingabezeile
\          vor CRTL-H verhindert es Loeschfunktion
ESC        Ende der Eingabe, Rueckkehr zum Kommandomodus
CRTL-?     Interrupt, beendet Eingabe
CRTL-D     Backtab, ueber autoindent
^CRTL-D    Loescht autoindent nur fuer eine Zeile
0CRTL-D    Loescht alle autoindent (autom. Einrueckung)
CRTL-V     nichtdruckbare Zeichen koennen editiert werden,
           wenn CRTL-V vorangestellt wird
```

- Eingabe und Ersetzen:

```
a          Anhaengen nach Kursor
i          Einfuegen vor Kursor
A          Anhaengen am Ende der Zeile
I          Einfuegen vor dem ersten "non-blank-Zeichen"
o          Eroeffnen Zeile (Eingabe von Textzeilen nach
           der aktuellen Zeile)
O          Eingabe von Textzeilen vor der aktuellen Zeile
rx         Ersetzen eines Zeichens durch das nach r
           angegebene Zeichen
Rstring    Ersetzen durch angegebene Zeichenkette
```

- Operatoren:

```
d          Streichen
c          Aendern von Woertern
<          Linksverschiebung einer Zeile um 1 Zeichen
>          Rechtsverschiebung einer Zeile um 1 Zeichen
!          Ausfuehren eines Kommandos vom Editor aus
=          Einruecken fuer LISP
y          Herausloesen einer Zeile und Speichern in einem
           Puffer
```

- Sonstige Operatoren:

```
C          Aendern des Restes der Zeile
D          Streichen bis Zeilenende
s          Ersetzen von Zeichen
S          Ersetzen von Zeilen
J          Verbinden von Zeilen
x          Streichen des Zeichens auf Kursorposition
X          Streichen des Zeichens vor Kursorposition
Y          Herausloesen von Zeilen
```

Herausloesen und Wiedereinbinden:

```
p          Zurueckspeichern von Zeilen nach der aktuellen
           Zeile (die durch y, Y herausgeloest wurden)
P          analog p, aber vor der aktuellen Zeile
xp         Zurueckspeichern aus dem durch x gekennzeichneten
           Puffer
"xd        Streichen in dem durch x gekennzeichneten Puffer
"xy        Herausloesen aus dem durch x gekennzeichneten
           Puffer
```

- Rueckgaengigmachen, Wiederholen, Wiederbekommen:

```
u          letzte Aenderung rueckgaengig machen
U          alle in der Zeile getaetigten Aenderungen rueck-
           gaengig machen
"dp        Wiederholen der letzten Streichungen
```

- Aufruf und Verlassen von vi:

```
%vi Datei
           Editiere die angegebene Datei, Kursor steht am
           Anfang der Datei
ZZ         Austritt aus vi, retten der durchgefuehrten
           Aenderungen
```

- Das Display:

```
letzte Zeile   enthaelt Fehlernachrichten, Eingaben zu den
               Operatoren :, /, ? und ! werden dort ange-
               zeigt, ebenso wie Meldungen ueber Ein-/Aus-
               gaben und groessere Aenderungen
@Zeilen    nur auf dem Bildschirm, nicht in der Datei
           (bei einfachen Terminals)
~Zeilen    Zeilen ueber das Dateiende hinaus
CRTL-x     Control-Zeichen, CRTL-? entspricht delete
tabs       werden als Leerzeichen expandiert, der Kursor
           steht auf dem letzten Zeichen
```

- Beispiele fuer einfache Kommandos:

```
dw         Streichen eines Wortes
de         Streichen eines Wortes, Punkt. bleibt erhalten
dd         Streichen einer Zeile
3dd        Streichen von drei Zeilen
i#Text#ESC  Einfuegen von Text
cw#Neu#ESC  Aendern eines Wortes in angegebenes neues Wort
```

C O M M

WEGA-Kommunikation

Vorwort

Dieser Artikel beschreibt das WEGA-Kommunikationspaket zur Kommunikation zwischwn WEGA und In-Circuit-Emulatoren ueber einen seriellen Kanal.
Das WEGA-Kommunikationspaket umfasst die folgenden Kommandos:

LOAD	uebertraegt WEGA-Dateien (a.out) in den Speicher des Emulators.
SEND(U8SEND)	legt einen Speicherbereich des Emulators als Datei (a.out) in das WEGA-Dateisystem ab.

An dieser Stelle sei auch auf die entsprechenden 'man-Eintraege' hingewiesen.

Weitere Moglichkeiten der Dateiuebertragung sind unter

WEGA-REMOTE
Benutzerhandbuch
PUTFILE, ...

und

WEGA-Dienstprogramme(B)
UUCP
Implementierung

zu finden. Diese Pakete geben die Moeglichkeit der Dateiuebertragung zwischen zwei WEGA-Systemen bzw. zwischen einem 'Remote System' (WEGA-System) und einem System auf dem entsprechend angepasste Remote-Software (z.B. unter dem Betriebssystem UDOS) laeuft.

Inhaltsverzeichnis Seite

1. Einfuehrung

Das WEGA-Kommunikationspaket LOAD/SEND(U8SEND) ermoeglicht unter der Steuerung von WEGA den Transfer von Anwenderprogrammen zwischen dem P8000 und einem Entwicklungsmodul (In-Circuit-Emulator) der Mikroprossesorfamilien U881/882 (Einchipmikrorechner) bzw. U8000 (16-Bit-Mikroprossesorsystem) ueber einen seriellen V.24-Kanal in beiden Richtungen. Der entsprechende Emulator ist anstelle eines Bildschirmterminals an einen TTY-Steckverbinder am P8000-Grundgeraet anzuschliessen. Das fuer den Dialog mit dem Emulator bzw. WEGA notwendige Bildschirmterminal wird dann entsprechend der Betriebsvorschrift des Emulators direkt mit diesem verbunden. Der eigentliche Datentransfer zwischen dem P8000 und dem jeweils angeschlossenen Emulator erfolgt auf der Basis des Standard-T-Formats.
Zu beachten ist, dass der angeschlossene Emulator ueber eine Software (Firmware) verfuegt, die den Festlegungen des Standard-T-Formats gerecht wird und besonders der speziellen Auslegung der Kommandos LOAD/SEND(U8SEND).

2. Datentransfer P8000 ---> Emulator (LOAD)

Das LOAD-Kommando uebertraegt ein U881/882 bzw. U8000 Programm aus dem Wega -Dateisystem in einen entsprechenden an das P8000 angeschlossenen In-Circuit Emulator. Die Datei mit dem Namen 'dateiname' muss das Format a.out (Lademodulformat) haben, ausfuehrbar sein und vom Typ N_MAGIC1, X_MAGIC1 oder X_MAGIC3 sein. LOAD ermittelt aus dem Kopf des Lademoduls die Ladepunkte der zu uebertragenden Text- und Datenbereiche. Die Segmenttabelle enthaelt die oberen acht Bits von jedem Ladepunkt. Die Text-, Daten- und BSS-Sektionen werden dabei immer auf volle 256-Bytegrenzen geladen.
Sollen Programme fuer den Einchiprechner U881/882 in den dafuer entsprechenden Emulator geladen werden, gelten folgende Festlegungen:

Kombinierter Code- und Datenbereich:
Es werden der Reihenfolge nach der Code- und anschliessend der Datenbereich in den Emulator geladen. Die Datensektion wird als Teil des Programmspeichers betrachtet.

Getrennter Code- und Datenbereich:
Es wird nur der Codebereich geladen. Der Datenbereich bezieht sich auf den externen Speicher.

BSS-Bereich:
Der BSS-Bereich des U881/U882 bezieht sich auf die REGISTER-Sektion; er wird nicht geladen.

Die binaeren Daten der Datei werden in das Standard-T-Format konvertiert und satzweise zum Emulator gesendet. Dort wird der Satz auf Gueltigkeit geprueft und an

das P8000 eine entsprechende Empfangsbestaetigung geschickt. Sollte diese die Ungueltigkeit des letzten Satzes bestaetigen, dann wird durch wiederholtes Senden versucht, eine gueltige Uebertragung zu erreichen. Gelingt das nicht, wird ein Satz mit einer Fehlermeldung gesendet und LOAD abgebrochen.

3. Datentransfer Emulator ---> P8000 (SEND(U8SEND))

Das SEND-Kommando uebertraegt einen naeher spezifizierten Speicherbereich eines entsprechenden an das P8000 angeschlossenen In-Circuit-Emulators in das WEGA-Dateisystem. Der ausgelagerte Speicherbereich wird als ein komplettes U881/882 bzw. U8000-Programm betrachtet. Das SEND-Programm eroeffnet eine Datei mit dem im Kommando notierten Namen 'dateiname' und sendet an den Emulator eine entsprechende Bestaetigung, dass der Datentransfer beginnen kann. Der Inhalt des Emulatorspeichers wird dann satzweise auf der Basis des Standard-T-Formats zum P8000 gesendet. Jeder Satz wird vom P8000 auf Gueltigkeit bzw. Ungueltigkeit ueberprueft und eine entsprechende Empfangsbestaetigung an das P8000 geschickt. Sollte es nicht gelingen, die Datei zu eroeffnen oder den Datentransfer exakt durchzufuehren, dann wird an den Emulator eine Fehlermeldung gesendet und SEND abgebrochen. Die erstellte Datei hat das Format a.out (Lademodulformat), ist ausfuehrbar (das s_flag-Byte hat das 'SF_SEND'-Bit gesetzt) und ist vom Typ X_MAGIC1 bzw. N_MAGIC1. Sollte schon eine Datei mit dem selben Namen existieren, dann wird diese ueberschrieben. Die im Kopf des Lademoduls angegebene Textgroesse des Lademoduls ist die Groesse des gesamten ausgelagerten Programms. Es besteht keine Moeglichkeit, Daten- und BSS-Abschnitte zu trennen. Die Datei kann erneut in einem Emulator mit dem Kommando LOAD geladen werden, dort beliebig bearbeitet werden und mit SEND ausgelagert werden.

4. Das Standard-T-Format

Das zu transferierende Anwenderprogramm, das als binaere Datei vorhanden sein muss, wird prinzipiell nach folgendem Schema konvertiert.

- Jeweils max. 32 Bytes der zu transferierenden Datei werden zu einem Satz zusammengefasst, mit zusaetzlichen Informationen versehen und dann seriell uebertragen.

- Sollte eine Datei kuerzer als 32 Byte bzw. nicht eine Laenge des Vielfachen von 32 Byte haben, dann kann der einzige bzw. letzte Satz auch kuerzer sein.

- Der Abschluss des Datentransfer ist immer ein spezieller "Endesatz"

mit der Laenge von acht Bytes.

- Die zu transferierenden Saetze werden jeweils von der empfangenen Seite, also beim LOAD-Kommando vom Emulator, beim SEND(U8SEND)-Kommando vom P8000 auf die Richtigkeit der Uebertragung anhand der zusaetzlichen Informationen ueberprueft. Sollten Fehler festgestellt werden, wird im begrenzten Umfang versucht, die Uebertragung solange zu wiederholen, bis sie fehlerfrei uebertragen wurde. Sollte auch das nicht gelingen, dann werden auf dem Bildschirmterminal entsprechende Fehlernachrichten dem Anwender mitgeteilt.

4.1. Datensatz

Fuer den Satz 1 ... n-1 gilt:

Satz-marke	Lade-adresse	Satz-laenge	Pruef-summe1	Datei-daten	Pruef-summe2	<CR>
/	aaaa	##	ss	dd..dd	cc	

/ Kennzeichen fuer den Anfang des Satzes.

aaaa Diese vier ASCII-Zeichen repraesentieren eine hexadezimale Adresse, ab der das erste Byte des Satzes in den Speicher des Emulators geladen wird. Die weiteren Bytes werden auf die darauf folgenden Speicherplaetze geschrieben.

Diese zwei ASCII-Zeichen geben die hexadezimale Anzahl der Bytes des Satzes an .Es koennen max. 32 Bytes sein.

ss Diese zwei ACCII-Zeichen sind die hexadezimale Summe der ASCII-Zeichen der Ladeadresse und der Satzlaenge .

dd...dd Das sind jeweils zwei ACCII-Zeichen pro Datenbytes der zu transferierenden Datei. Es sind max. 32 Bytes moeglich.

cc Diese zwei ASCII-Zeichen sind die hexadezimale Summe modulo 256 der ASCII-Zeichen der Datenbytes .

<CR> Satzende

4.2. Endesatz

Fuer den Satz n (Endesatz) gilt:

Satz-marke /	Lade-adresse aaaa	Satz-laenge ##	Pruef-summe ss	<CR>

/ Kennzeichen fuer den Anfang des Satzes

aaaa 0000H <--- bei LOAD Ladeadresse

00H

ss 00H <--- bei LOAD Pruefsumme

<CR> Satzende

4.3. Empfangsbestaetigungen

Neben der eigentlichen Uebertragung von Datensaetzen, werden noch Informationen zwischen dem P8000 und dem entsprechenden Emulator ausgetauscht, die folgendes signalisieren:

0<CR> ACK-Sequenz: Es kann ein Datensatz gesendet werden bzw es wird bestaetigt, dass die Pruefsummen des gerade empfangenen Datensatzes gueltig sind und die Bereitschaft zum Empfang eines neuen Satzes gegeben ist.

7<CR> NAK-Sequenz: Die Pruefsummen des gerade empfangenen Datensatzes sind ungueltig; d.h. dieser Satz ist nochmal zu senden. Wenn nach der 10. Wiederholung immer noch eine unkorrekte Ubertragung vorliegt, dann erfolgt ein genereller Abbruch derDatenuebertragung.

9<CR> Einleitung einer Fehlernachricht: Die vorangegangene Datenuebertragung, ACK/NAK-Sequenz oder die Kommandoparameter sind derart unzulaessig, dass ein Programmabbruch notwendig wird.

5. Kommandoaufruf

Wie bereits im Abschn.1 erwaehnt, wird der Emulator anstelle eines Bildschirmterminals an einen TTY-Steckverbinder am P8000-Grundgeraet angeschlossen. Das Terminal wird dann direkt mit dem Emulator verbunden. Saemtliche Eingaben des Bedieners, also auch solche, die ausschliesslich

fuer das WEGA-System bestimmt sind, muessen damit zwangsweise den Emulator durchlaufen. Daraus folgt, dass die Bedienereingabe (Kommando) moeglicherweise von der Emulatorfirmware modifiziert werden kann. Aus der Betriebsvorschrift des jeweiligen Emulators ist zu entnehmen, ob auch tatsaechlich das, was der Bediener entsprechend der geforderten Kommandosyntax eingibt, an das WEGA gelangt. Werden durch die Emulatorfirmware Modifikationen vorgenommen, so ist es im Interesse einer problemlosen Kommandoabarbeitung zweckmaessig, diese mit der Hilfe derC-Shell-Funktion

    alias

zu korrigieren.

Beispiel:

Die Emulatorfirmware setzt vor die Bedienereingabe ein B; . An das WEGA wuerde dann

    B;'bedienereingabe'

gelangen. D.h. B; muss unterdrueckt werden. Dazu ist vor der eigentlichen Kommandoeingabe

    alias B ;

zu geben. Dadurch wird das vom Emulator gesendete

    B;'bedienereingabe' zu ;;'bedienereingabe'

transformiert. (;; sind zwei leere Kommandos). Damit diese alias-Funktion nicht nach jedem 'login' eingegeben werden muss, kann sie in die '.cshrc'-Datei eingetragen werden.

5.1. Kommandoaufruf LOAD

Die Syntax fuer dieses Kommando lautet:
    LOAD 'dateiname'

Es wird der Inhalt der Datei aus dem WEGA-Dateisystem mit dem Namen 'dateiname' in den Speicher des Emulators transferiert. Es ist gleichgueltig, ob es sich dabei um einen P8000- oder U881/882-Emulator handelt. Fuer die beim Kommando LOAD notierte Datei mit dem Namen 'dateiname' gilt das bereits im Abschn.2 Beschriebene, also:

    Format   : a.out (Lademodulformat)
    Ausfuehrbarkeit : ja
    Typ   : N_MAGIC1, X_MAGIC1 oder N_MAGIC1

Ausfuehrung:

Der Emulator sendet die Kommandozeile zu WEGA, was den eigentlichen Start des LOAD-Programms bewirkt. Das so gestartete WEGA-Programm LOAD eroeffnet die Datei mit dem Namen 'dateiname', und nach Ueberpruefung der Gueltigkeit der oben geforderten Dateiparameter wird der Datentransfer durchgefuehrt. Die Gueltigkeit einer vollstaendigen Uebertragung wird durch das Prompt-Zeichen des Emulator-Monitorprogramms signalisiert.

Ergebnis:

Im Speicher des Emulators ist dann ein abarbeitungsfaehiges Maschinenprogramm vorhanden. Dieser Speicherbereich bzw. auch ein bearbeiteter kann mit dem Kommando SEND(U8SEND) wieder in das WEGA-Dateisystem transferiert werden und erneut mit dem Kommando LOAD in den Emulator geladen werden.

5.1.1. Beispiel LOAD

Es ist die folgende Datei mit dem Dateinamen OTTO aus dem WEGA-Dateisysem des P8000 in einen In-Circuit-Emulator zu transferieren. Die Datei besteht aus den sechs Datenbytes:

ED 5E F3 C3 00 08

und hat die Startadresse 800H.

Der Start der gewuenschten Prozedur ist gemaess des Abschn.2. bzw. 5.1. vorzunehmen; also:

```
alias B;
LOAD OTTO
```

Wenn die Datei das Format a.out hat, ausfuehrbar ist und vom entsprechenden Typ (N_MAGIC1, X_MAGIC1 oder X_MAGIC3) ist, wird sich folgender Datenstrom zwischen dem P8000 und dem Emulator (EM) ergeben. Das Endergebnis ist dann, dass die oben angegebenen sechs Datenbytes im Speicher des Emulators stehen.

Es wird sich der fogende Datenstrom ergeben:

```
        /      Satzmarke       1. (und einziger) Datensatz
        0      Ladeadresse
        8      .
        0      .
        0      .
        0      Satzlaenge
        6      .
        0      Pruefsumme1     0+8+0+0+0+6=0E
        E      .
        E      Datenbytes
        D      .
        5      .
---->   E      .
 EM     F      .
        3      .
        C      .
        3      .
        0      .
        0      .
        0      .
        8      .
        5      Pruefsumme2     E+D+5+E+F+3+C+3+0+0+0+8=57
        7      .
       <CR>
<----   7      NAK-Sequenz     Pruefsumme1 oder/und 2 des
P8000  <CR>                    Datensatzes ist ungueltig
        /      Satzmarke       Wiederholung des Datensatzes
        0      Ladeadresse
        8      .
---->   .      .
 EM     .      .
        .      .
        7      .
       <CR>
<----   0      ACK-Sequenz     Pruefsumme1 und 2 des Daten-
P8000  <CR>                    satzes ist gueltig
        /      Satzmarke       Endesatz
        0      Ladeadresse
        8      .
        0      .
---->   0      .
 EM     0      Satzlaenge
        0      .
        0      Pruefsumme      0+8+0+0=8
        8      .
       <CR>
<----   0      ACK-Sequenz     Pruefsumme des Endesatzes
P8000  <CR>                    ist gueltig
```

5.2. Kommandos SEND(U8SEND)

Das WEGA-Kommunikationspaket verfuegt ueber die folgenden Kommandos, die es gestatten, einen naeher spezifizierten Emulatorspeicherbereich in das WEGA-Dateisystem auszu-lagern:

SEND Auslagern eines Speicherbereiches des 16-Bit-Mikroprossesoremulators-U8000 in das P8000.

U8SEND Auslagern eines Speicherbereiches des Einchipmikrorechneremulators-U881/882 in das P8000.

Die Voraussetzung fuer die Anwendung dieser Kommandos ist, dass der jeweils typgerechte Emulator an das P8000 angeschlossen ist.

5.2.1. Kommandoaufruf SEND

Die Syntax fuer dieses Kommando lautet:

```
SEND 'dateiname' beginnadr endadr [startadr]
```

Es wird der durch die Angabe der Beginnadresse (beginnadr) und der Endadresse (endadr) als hexadezimale Zahlen ohne fuehrende Ox spezifizierte Emulatorspeicherbereich in das WEGA-Dateisystem in die Datei mit dem Namen 'dateiname' ue-bertragen. Sollte schon eine Datei mit dem gleichen Namen existieren, dann wird diese ueberschrieben. Die Starta-dresse (startadr) ist optional. Der Standardwert fuer die Startadresse ist die Beginnadresse. Wird eine Startadresse notiert, dann muss sie auf einer 256-Bytegrenze liegen.

Ausfuehrung:

Der Emulator sendet die Kommandozeile zu WEGA, was den eigentlichen Start des SEND-Programms bewirkt. SEND eroeffnet die Datei mit dem Namen 'dateiname' und speichert in ihr die vom Emulator empfangenen Daten ab.
Die Gueltigkeit einer vollstaendigen Uebertragung wird durch das Prompt-Zeichen des Emulator-Monitorprogramms sig-nalisiert.
Die Startadresse wird in die WEGA-Datei eingetragen und das SEND-Programm beendet.

Ergebnis:

Der ausgelagerte Speicherbereich wird als ein komplettes U8000-Programm betrachtet. Die erstellte Datei hat das Lademodulformat a.out und ist vom Typ N_MAGIC1. Sie ist ausfuehrbar, da im s_flag-Byte das SF_SEND-Bit (Ox0010) gesetzt ist.
Die im Kopf des Lademoduls angegebene Textgroesse des Lademoduls

ist die Groesse des gesamten ausgelagerten Programms. Es besteht keine Moeglichkeit, Daten- und BSS-Abschnitte zu trennen.
Die Datei kann erneut in den Emulator mit dem Kommando LOAD geladen , dort beliebig bearbeitet und mit SEND ausgelagert werden.

5.2.2. Kommandoaufruf U8SEND

Die Syntax fuer dieses Kommando lautet:

```
        SEND 'dateiname' beginnadr n-1 [startadr]
```

Es wird der durch die Angabe der Beginnadresse (beginadr) und der Anzahl der zu transferierenden Bytes minus eins (n-1) als hexadezimale Zahlen ohne fuehrende Ox spezifizierten Emulatorspeicherbereich in das WEGA-Dateisystem in die Datei mit dem Namen 'dateiname' uebertragen. Sollte schon eine Datei mit dem gleichen Namen existieren, dann wird diese ueberschrieben. Die Startadresse (startadr) ist optional. Der Standardwert fuer die Startadresse ist die Beginnadresse. Wird eine Startadresse notiert, dann muss sie auf einer 256=Bytegrenze liegen.

Ausfuehrung:

Der Emulator sendet die Kommandozeile zu WEGA. Da dieses Auslagerungsprogramm jedoch unter dem Namen U8SEND im WEGA-Dateisystem gespeichert ist, muss es demzufolge auch durch ein entsprechendes Kommando aufgerufen werden.
Aus der Betriebsvorschrift des Emulators ist zu entnehmen, ob auch tatsaechlich U8SEND zum P8000 gesendet wird. Ist das nicht der Fall, dann ist SEND vor Eingabe des Kommandos mit Hilfe der alias-Funktion in U8SEND zu transformieren, also:

```
        alias SEND U8SEND
```

Wird diese Aktivitaet unterlassen, dann wird das SEND-Programm (U8000-Auslagerungsprogramm) gestartet.
Soll nach dem gleichen 'login' ein Programm aus einem U8000-Emulator ausgelagert werden, so muss diese Kommandotranformation durch

```
        unalias SEND
```

rueckgaengig gemacht werden. Aus diesen Grund sollte diese alias-Funktion auch nicht in die '.cshrc'-Datei eingetragen werden.
U8SEND eroeffnet die Datei mit dem Namen 'dateiname' und speichert in ihr die vom Emulator empfangenen Daten ab.
Die Gueltigkeit einer vollstaendigen Uebertragung wird durch das Prompt-Zeichen des Emulator-Monitorprogramms signalisiert.
Die Startadresse wird in die WEGA-Datei eingetragen und das U8SEND-Programm beendet.

Ergebnis

Der ausgelagerte Speicherbereich wird als ein kommplettes U881/882-Programm betrachtet. Die erstellte Datei hat das Lademodulformat a.out und ist vom Typ X_MAGIC1. Sie ist ausfuehrbar, da im s_flag-Byte das SF_SEND-Bit (Ox0010) gesetzt ist.
Die im Kopf des Lademoduls angegebene Textgroesse des Lademoduls ist die Groesse des gesamten ausgelagerten Programms. Es besteht keine Moeglichkeit, Daten- und BSS-Abschnitte zu trennen.
Die Datei kann erneut in den Emulator mit dem Kommando LOAD geladen, dort beliebig bearbeitet und mit U8SEND ausgelagert werden.

5.2.3. Beispiel SEND(U8SEND)

Es ist der aus den folgenden sechs Bytes bestehende Speicherbereich aus einem Einchipmikrorechneremulator-U881/882 in das WEGA-Dateisystem in die Datei mit dem Namen HUGO auszulagern.

```
ED 5E F3 C3  00 08
```

Die Beginnadresse des Speicherbereiches hat den hexadezimalen Wert 800.

Der Start der gewuenschten Prozedur ist gemaess des Abschn.3. bzw. 5.2.2.

```
alias SEND U8SEND SEND 800 5
```

```
---->   0       ACK-Sequenz      bereit zum Empfang eines Da-
 EM    <CR>                      tensatzes
        /       Satzmarke        1. (und einziger) Datensatz
        0       Ladeadresse
        8       .
        0       .
        0       .
        0       Satzlaenge
        6       .
        0       Pruefsumme1      0+8+0+0+0+6=0E
        E       .
        E       Datenbytes
        D       .
<----   5       .
P8000   E       .
        F       .
        3       .
        C       .
        3       .
        0       .
        0       .
        0       .
        8       .
        5       Pruefsumme2      E+D+5+E+F+3+c+3+0+0+0+8=57
        7       .
       <CR>
---->   0       ACK-Sequenz      Pruefsumme1 und 2 des Daten-
 EM    <CR>                      satzes ist gueltig
        /       Satzmarke        Endesatz
        0
        0
        0
<----   0
P8000   0
        0
        0
        0
       <CR>
```

6. Fehlernachrichten

Nachdem eine Fehlernachricht durch die Sequenz 9<CR> angekuendigt wurde, folgt darauf die durch // eingeleitete Fehlernachricht.
Folgende Fehlernachrichten sind moeglich:

//ABORT -7-	Datensatz nach 10-maliger Wiederholung immer noch nich korrekt uebertragen.
//ABORT -x-	Fehlerhafte Uebertragung der ACK/NAK-Sequenz. Anstelle 0<CR>/7<CR> wurde x<CR> uebertragen. x ist ein von 0/7 verschiedenes ASCII-Zeichen.
//bad ack sequence	Fehlerhafte ACK/NAK-Sequenz. Anstelle von 0<CR>/7<CR> wurden mehrere andere ASCII-Zeichen uebertragen.
//can't open file (null)	Beim Kommando LOAD wurde kein Dateiname notiert.
//can't open file dateiname	Die beim Kommando LOAD notierte Datei mit dem Namen 'dateiname' existiert nicht.
//illegal file type	Die beim Kommando LOAD notierte Datei mit dem Namen 'dateiname' hat nicht das Format a.out.
//file read error	Die beim Kommando LOAD notierte Datei mit dem Namen 'dateiname' hat das Format a.out, ist jedoch fehlerbehaftet.
//'dateiname' not executable	Die beim Kommando LOAD notierte Datei mit dem Namen 'dateiname' ist nicht ausfuehrbar.
//entry point too low	Die Startadresse liegt vor dem uebertragenen Speicherbereich.

Zu beachten ist ,dass, da das Bildschirmterminal entsprechend

Abschn.1. direkt mit dem Emulator verbunden ist, nach der Fehlernachrichtsequenz 9<CR> von der Emulator-Firmware weitere Nachrichten kreiert werden koennen. Diese sind der Betriebsvorschrift des entsprechenden Emulators zu entnehmen.

U 8 0 0 0 - P L Z / A S M

Benutzerhandbuch

Vorwort

Diese Beschreibung gibt einen Ueberblick ueber die Handhabung des PLZ/ASM-Sprachuebersetzers (as) unter dem Betriebssystem WEGA. In dieser Unterlage werden implementationsabhaengige Eigenschaften beschrieben. Die vorliegende Version von PLZ/ASM haengt von einigen Eigenschaften des Betriebssystems WEGA ab. Sie benutzt zur Dateiarbeit das Ein-/Ausgabepaket des Systems.

Eine Beschreibung fuer den exakten Aufruf ist unter as(1) im WEGA-Programmierhandbuch zu finden. Erlaeuterungen zum Maschinenkodeformat sind unter a.out(5) im WEGA-Programmierhandbuch nachzuschlagen. Hinweise zum Objektkodeformat werden im Abschn. 7 dieser Beschreibung gegeben.

Inhaltsverzeichnis Seite

1. Einleitung

1.1. Allgemeine Beschreibung

Der U8000-PLZ/ASM-Assembler (aufgerufen als Kommando as) ist ein verschieblicher Assembler fuer WEGA. Es wird eine Quelldatei (die symbolische Darstellung eines Programms in U8000-PLZ/ASM-Sprache) in ein Objektmodul uebersetzt. Gleichzeitig kann eine Listingdatei erzeugt werden, die die Quellzeilen und den uebersetzten Kode enthaelt.

1.2. Verschieblichkeit

Verschieblichkeit bedeutet die Moeglichkeit der Zuordnung eines Programms und seiner Daten auf einen bestimmten Speicherbereich im Anschluss an den Uebersetzungsprozess.
Die Ausgabe des Assemblers ist ein Objektmodul, der genug Informationen enthaelt, die es einem Linker oder Lader gestatten, diesen Modul einem Speicherbereich zuzuordnen. Weitere Hinweise zum WEGA-Linker sind unter ld(1) im WEGA-Programmierhandbuch enthalten.

1.3. Assembler-Abbruchbedingungen

Es gibt zwei Abbruchbedingungen fuer den Assembler.

(a) Falls Ein-/Ausgabefehler waehrend eines Systemaufrufs auftreten, so erfolgt eine Fehlerausgabe und der Assemblerlauf wird abgebrochen.

(b) Falls Systemfehlerbedingungen auftreten, die einen Abbruch erzwingen, so wird der Assembler-Abbruchfehler (Fehlernr. 255) an die Standardfehlerausgabe und die Listingdatei ausgegeben.

2. Ein-/Ausgabe

2.1. Benutzereingabe

Zum Erstellen von U8000-PLZ/ASM-Quellprogrammen wird ein Editor benutzt. Der Name der Quelldatei muss die Endung '.s' (gross oder klein geschrieben) besitzen. Hinweise zum Aufruf des Assemblers werden im Abschn. 3 gegeben.

2.2. Assembler-Ausgabe

Der Assembler erzeugt zwei Dateien: eine Listingdatei (implizit der Quelldateiname, aber mit der Endung '.l' an Stelle von '.s') und eine Objektdatei (mit a.out oder t.out als impliziten Dateinamen, s. Abschn. 7). Fuer die

Erzeugung der Objektdatei benutzt der Assembler eine temporaere Zwischendatei, die zum Abschluss des Assemblerlaufs gestrichen wird. Die Listingdatei enthaelt die Quellzeilen mit entsprechenden Zeilennummern. Fehlernachrichten werden nach der Zeile angegeben, in der der Fehler auftritt. Siehe Abschn. 8 fuer naehere Erlaeuterungen zu den Fehlernummern.

3. Assembler-Kommandozeile

3.1. Kommandozeile

Der Assembler wird durch Eingabe der Kommandozeile

```
as Dateiname [Optionen]
```

aufgerufen. Die Endung '.s', die die Datei als Quelltextdatei fuer einen U8000-PLZ/ASM-Modul kennzeichnet, muss bei der Angabe des Dateinamens mit angegeben werden.

3.2. Optionen

Die folgenden Optionen koennen in beliebiger Reihenfolge, getrennt durch Leer- oder Tabulatorzeichen, angegeben werden:

-d string — Die angegebene Zeichenkette (max. 19 Zeichen) wird in den Listingkopf uebernommen. Gilt nur in Verbindung mit der Option -l. Wird meist benutzt, um das Datum anzugeben.

-f — Gestattet die Uebersetzung von Gleitkomma-anweisungen.

-i — Die temporaere Zwischendatei wird nach Abschluss des Assemblerlaufs nicht gestrichen. Sie erhaelt den Namen der Quelldatei, aber mit der Endung '.i'.

-l — Bewirkt die Erzeugung einer Listingdatei. Sie erhaelt den Namen der Quelldatei, aber mit der Endung '.l'. Wenn diese Option nicht angegeben ist, so wird keine Listingdatei erzeugt.

-o filename — Erlaubt den Nutzer die Angabe eines Dateinamens fuer die erzeugte Objektkodedatei.

-p — Bewirkt die Ausgabe der vollstaendigen Listingdatei ueber die Konsole. Wenn diese Option nicht angegeben ist, so werden nur die Quellzeilen, die einen Fehler

enthalten, ueber die Konsole ausgegeben.

-r Die Datei mit den Informationen wird nach Abschluss des Assemblerlaufs nicht gestrichen. Sie erhaelt den Namen der Quelldatei, aber mit der Endung '.r'.

-u Alle nichtdefinierten Symbole werden als externe Groessen behandelt.

-v Schaltet zusaetzliche Konsolausgaben, wie Name, Versionsnummer, Pass1-Meldung und Abschlussmeldung ein.

-z Veranlasst den Assembler an Stelle des zu erzeugen. Der Dateiname ist dann t.out an Stelle von a.out (s. Abschn. 7).

-^ Schaltet die Pass1-Trace-Moeglichkeit ein.

4. Listingformat

4.1. Formatbeschreibung

Der Assembler erzeugt ein Listing des Quellprogramms, das den erzeugten Objektkode enthaelt. Die verschiedenen Felder des Listingformats werden in diesem Abschnitt beschrieben. Der folgende Abschnitt 4.2. illustriert an einem Beispielprogramm den Aufbau des Listings.

HEADING Die erste Seite enthaelt einen Kopf, in dem die Versionsnummer und die nachfolgend beschriebenen Spaltenueberschriften erscheinen. Zusaetzlich kann noch eine vom Nutzer ueber die Option -d festgelegte Zeichenkette (meist fuer das Datum benutzt) enthalten sein.

LOC In dieser Spalte werden die relativen Adressen der Anweisungen angegeben. Dieser Zaehler startet fuer jede unterschiedliche Sektion bei Null.

OBJ CODE Diese Spalte enthaelt die Werte fuer den generierten Objektkode. Sie bleibt fuer Anweisungen, die keinen Objektkode generieren, leer. Im Anschluss an jedes Byte oder Wort des Objektkodes folgt entweder ein Apostroph ('), ein Stern (*) oder ein Leerzeichen. Ein Apostroph zeigt an, dass es sich um einen verschieblichen Wert handelt. Ein Stern besagt, dass der Wert von einem externen Symbol abhaengt. Wenn ein Leerzeichen folgt, so aendert sich der Wert nicht mehr. Ein

verschieblicher oder von einer externen Groesse abhaengiger Wert wird durch den Linker modifiziert. Der Wert waehrend des Programmlaufs kann sich dann vom Wert im Listing unterscheiden. Drei Punkte (...) kennzeichnen nur bei Dateninitialisierungen eine Wiederholung des vorherigen Bytes, Wortes oder Long-Wortes.

STMT — Diese Spalte enthaelt die fortlaufenden Nummern der Quellzeilen.

SOURCE — Der Rest der Zeile enthaelt den Quelltext.

4.2. Beispiel-Listing

```
plzasm    1.6
LOC    OBJ CODE    STMT SOURCE STATEMENT
                      1 bubble_sort MODULE  ! Modulvereinbarung !
                      2
                      3  CONSTANT           ! Konstantenvereinbarungen !
                      4   FALSE := 0
                      5   TRUE  := 1
                      6
                      7  EXTERNAL
                      8   list ARRAY[10 WORD] ! zu sortierendes Feld !
                      9
                     10  INTERNAL
0000                 11   switch BYTE       ! Austauschanzeiger !
                     12
0000                 13   sort PROCEDUR     ! Prozedurvereinbarung !
                     14   ENTRY             ! Beginn ausfuehrbarer Teil !
                     15    DO
0000 4C05  0000'     17     LDB switch,#FALSE ! Initialisierung switch !
0004 0000
0006 8D18            17     CLR R1          ! Loeschen Feldzeiger i  !
                     18     DO
0008 8B01            19       CP R1,R0      ! Feldende erreicht !
000A E701            20       IF UGE THEN EXIT FI
000C E811
000E A112            21       LD R2,R1      ! Initialisierung Zeiger j !
0010 A921            22       INC R2,#2     ! j=i+1 (doppelt da Worte) !
0012 6114  0000*     23       LD R4,list(R1)
0017 6126  0000*     24       LD R6,list(R2)
001A 8B64            25       CP R4,R6      ! wenn list[i]>list[j] !
001C E307            26       IF UGT THEN   ! Austausch !
001E 4C05  0000'     27         LDB switch,#TRUE
0022 0101
0024 6F17  0000*     28         LD list(R1),R6
0028 6F24  0000*     29         LD list(R2),R4
                     30       FI
002C A911            31       INC R1,#2     ! naechste Element (Wort) !
002E E8EC            32     OD              ! Ende innere DO-Schleife !
0030 4C01  0000'     33     CPB switch,#FALSE ! war Austausch !
0034 0000
0036 EE01            34     IF EQ THEN RET FI
0038 9E08
003A E8E2            35   OD
                     36
003C                 37  END sort           ! Ende der Prozedur sort !
                     38
                     39  GLOBAL             ! neue Prozedurvereinbarung !
                     40
003C                 41  main PROCEDURE     ! Hauptprogramm !
                     42  ENTRY
003C 2100  0012      43   LD R0,#9*2        ! Feldlaenge !
0040 D021            44   CALR sort         ! Aufruf von sort !
0042 9E08            45   RET
0044                 46  END main
                     47 END bubble_sort
```

5. Minimale Programmanforderungen

Die Beispiele in diesem Abschnitt illustrieren die minimalen Anforderungen der PLZ/ASM-Struktur an ein Quellprogramm. Das erste Beispiel zeigt die absolut notwendigen minimalsten Strukturanforderungen: eine Moduldefinition, eine Festlegung der Klasse und eine Prozedurdefinition.
Das zweite Beispiel stellt dasselbe Programm, aber unter Verwendung von symbolischen Konstanten und Datenvereinbarungen, dar.

Beispiel 1:

```
anyname MODULE

 GLOBAL   ! oder INTERNAL je nachdem, ob !
          ! Modulverbindung noetig ist   !

 somename PROCEDURE
 ENTRY
    ! Das Programm beginnt hier !
    RET
 END somename
END anyname
```

Beispiel 2:

```
anyname MODULE

 CONST! hier werden symbolische Konstanten !
          ! vereinbart !
    one := 1
    hexten := %10

 GLOBAL     ! oder INTERNAL je nachdem, ob !
            ! Modulverbindung noetig ist   !

    a BYTE  ! Datenvereinbarungen koennen hier !
    b WORD  ! erscheinen !
    buffer ARRAY [100 BYTE]

 GLOBAL     ! Vereinbarungsklasse festlegen !
! hier redundant, da vorher schon !
! GLOBAL festgelegt wurde !

 somename PROCEDURE
 ENTRY
    ! Das Programm beginnt hier !
     RET
 END somename
END anyname
```

6. Implementationseigenschaften und Einschraenkungen

Die U8000-PLZ/ASM-Assemblereinschraenkungen und implementationsabhaengige Eigenschaften sind folgende:

1. Der U8000-PLZ/ASM-Assembler benutzt den Standard-ASCII-Zeichensatz. Gross- und Kleinbuchstaben werden unterschieden und als verschiedene Zeichen behandelt. Schluesselwoerter werden nur akzeptiert, wenn sie entweder vollstaendig gross oder klein geschrieben sind (z.B.: GLOBAL oder global, nicht aber Global). Hexadezimalzeichen und spezielle Zeichenketten koennen entweder gross oder klein geschrieben werden (z.B.: %Ab, '1st line %R2nd line%r').

2. Quellzeilen, die mehr als 132 Zeichen enthalten, werden akzeptiert; bei Fehlernachrichten werden aber nur 132 Zeichen ausgegeben. Kommentare und in Apostrophe eingeschlossene Zeichenketten koennen ueber mehrere Zeilen fortgesetzt werden. Beachtung sollte der Ausgewogenheit von oeffnenden und schliessenden Ausrufungszeichen (!) und Apostrophen (') geschenkt werden.

3. Zeichenketten koennen nicht die Laenge Null haben ('').

4. Konstanten werden intern als vorzeichenlose 32-Bit-Groessen repraesentiert. Jeder Operand in einem konstanten Ausdruck wird so berechnet, als waere er vom Typ LONG vereinbart. Zum Beispiel ist 4/2 gleich 2, aber 4/-2 ist gl;eich Null, da -2 als sehr grosse vorzeichenlose Zahl interpretiert wird. Waehrend der Berechnung von konstanten Ausdruecken erfolgt kein Test auf Ueberlauf. Wenn eine Zeichenfolge als Konstante benutzt wird, so sind nur die ersten vier Zeichen von Bedeutung, da Konstanten als 32-Bit-Werte repraesentiert werden ('ABCD'='ABCDE'). Eine Ausnahme dazu bildet eine Zeichenkette, die zur Feldinitialisierung benutzt wird. Diese kann bis zu 127 Zeichen haben.

5. Namen (Bezeichner) koennen aus maximal 127 Zeichen bestehen.

6. Nach dem Erkennen eines Fehlers innerhalb der Klassen CONSTANT und TYPE oder in einer Variablenvereinbarung springt der Assembler zum naechsten Schluesselwort, das eine neue Anweisung beginnt (ein Operationskode, IF, DO, EXIT, REPEAT oder END). Diese Vorgehensweise kann ein mehrmaliges Uebersetzen zum Erkennen und Beseitigen aller Fehler notwendig machen.

7. Objektkode

Der Assembler kann zwei unterschiedliche Objektkodeformate erzeugen:

- Objektkode, der kompatibel zum UDOS-PLZ/ASM-Objektkode (t.out) ist
- WEGA-Objektkode (a.out).

Siehe dazu Option -z im Abschn. 3.2.

Wenn WEGA-Objektkode erzeugt wird, so ist der implizite Dateiname a.out. Dieses Objektkodeformat wird vollstaendig unter a.out(5) im WEGA-Programmierhandbuch beschrieben.

8. PLZ/ASM-Fehlernachrichten

Die Fehlernachrichten entsprechen denen des PLZ/SYS-Compilers, siehe dazu U8000-PLZ/SYS Benutzerhandbuch, Abschn. 3.4.

U 8 0 0 0 - P L Z / S Y S

Benutzerhandbuch

Inhaltsverzeichnis Seite

1. Einfuehrung

1.1. Inhaltsuebersicht

Dieses Handbuch beschreibt, wie PLZ/SYS-Programme unter WEGA auf dem P8000 laufen. Enthalten sind Informationen ueber den Aufruf des Compilers "plzsys" und des Kodegenerators "plzcg" sowie Informationen ueber Programmausfuehrungsbedingungen und -vereinbarungen.
PLZ/SYS-Quellprogramme, die unter WEGA laufen, werden in Abschn. 2 behandelt, ebenfalls die Nutzung externer Nicht-PLZ/SYS-Prozeduren.
Der PLZ/SYS-Compiler wird in Abschn. 3 und der Kodegenerator in Abschn. 4 beschrieben.
Abschliessend wird in Abschn. 5 beschrieben, wie PLZ/SYS-Module, die unter dem Betriebssystem UDOS erstellt wurden und das PLZ-I/O-Paket benutzen, auf das WEGA-Betriebssystem uebertragen werden koennen.

1.2. Literaturhinweise

- Snook, Bass, Roberts, Nahapetian, Fay:
  Report on the Programming Language PLZ/SYS,
  Springer-Verlag, 1978

- U880-PLZ/SYS Benutzerhandbuch,
  Band "UDOS-Software, Programmiersprachen"
  der P8000-Dokumentation

- U8000-PLZ/ASM Benutzerhandbuch,
  Band "UDOS-Software, Mikroprozessorsoftware"
  der P8000-Dokumentation

- U8000 PLZ/ASM Benutzerhandbuch (as),
  Band "WEGA-Software, Dienstprogramme"
  der P8000-Dokumentation

- Kommandos plz(1), plzsys(1), plzcg(1)
  uimage(1), as(1), ld(1),
  Band "WEGA-Software, Programmierhandbuch"
  der P8000-Dokumentation

- Hoehere Programmiersprache fuer Echtzeitsteuerungen
  PLZRTC Vers. 1.0, Sprachbeschreibung,
  Technische Hochschule Karl-Marx-Stadt 1980

- PLZRTC Compiler 1.0, Bedienhandbuch,
  Technische Hochschule Karl-Marx-Stadt 1980

- Z80 - Echtzeitbetriebssystemkern
  Kodegenerator RTCCG, Beschreibung/Bedienhandbuch,
  Technische Hochschule Karl-Marx-Stadt 1980

- Classen, Oefler:
  Wissensspeicher Mikrorechnerprogrammierung,
  VEB Verlag Technik, Berlin 1986

1.3. PLZ/SYS-Uebersicht

Der PLZ/SYS-Compiler (plzsys), der Kodegenerator (plzcg) und der PLZ-Treiber (plz) sind in diesem Handbuch beschrieben.
In Verbindung mit einem WEGA-Editor koennen PLZ/SYS-Quellprogramme erstellt und dann zu einem verschieblichen Objektmodul verarbeitet werden, der unter WEGA lauffaehig oder in ein U8000-System ladbar ist.
Ein PLZ/SYS-Programm ist aus separat uebersetzten Quellmodulen zusammengesetzt. Ein PLZ/SYS-Quellmodul kann Steuerzeilen der Form

```
#include "filename"
```

enthalten. Solch eine Steuerzeile bewirkt, dass diese Zeile durch den Inhalt der Datei "filename" ersetzt wird.

Es gibt 4 Stufen bei der Verarbeitung eines PLZ/SYS-Moduls:

1. Verwendung von "cpp" (C-Preprozessor) zur Ersetzung aller Steuerzeilen in dem Quellmodul
2. Verwendung von "plzsys" zur Erstellung eines Zwischenkodemoduls (Z-Kode-Modul)
3. Verwendung von "plzcg" zur Erstellung eines Maschinenkodemoduls im zobj-Format
4. Verwendung von "uimage" zur Uebersetzung des Resultats von Schritt 3 in a.out-Format

Nachdem alle PLZ/SYS-Module eines Programms bearbeitet wurden, kann der WEGA-Linker (ld) benutzt werden, um alle diese Maschinenkodemodule sowie andere existierende Maschinenkodemodule (Bibliotheken (libraries), uebersetzte Assemblerprogramme, uebersetzte C-Programme) zu einem Objektmodul zusammenzubinden, der unter WEGA laufen kann (s. Bild 1-1).
Das PLZ-I/O-Paket ist in der Bibliothek "/usr/lib/libp.a" enthalten, ebenso eine PLZ-aufrufbare Version der WEGA-Systemaufrufe (s. Abschn. 5).
In diesem Benutzerhandbuch sind alle Dateinamenserweiterungen in Kleinbuchstaben geschrieben. Ebenso sind Grossbuchstaben erlaubt (z.B. .p, .P, .z, .Z).

```
-----------------
PLZ/SYS-Quelle
mit
C-Preprozessor-
steueranweisungen
-----------------
  I
  I cpp
  I
-----------------
PLZ/SYS-
   Zwischenquelle
(file.p)
-----------------
  I
  I plzsys
  I
-----------------   ----------------   ---------   ---------
Z-Kode-Objekt-      PLZ/ASM-Quelle     PLZ-I/O-     C-Bibl.
            modul                          Bibl.
(file.z)            (file.s)           (usr/lib/   (lib/
                                         libp.a)     libc.a)
-----------------   ----------------   ---------   ---------
  I                   I                    I           I
  I plzcg             I as                 I           I
  I                   I                    I           I
-----------------   ----------------       I           I
Maschinenkode-      Maschinenkode-         I           I
Objektmodul in      Objektmodul            I           I
      zobj-Format   in a.out-Format        I           I
(t.out (file.t))    (a.out (file.o))       I           I
-----------------   ----------------       I           I
  I                        I               I           I
  I uimage                 I               I           I
  I                        I               I           I
-----------------          I               I           I
Maschinenkode-             I               I           I
Objektmodul in             I               I           I
     a.out-Format          I               I           I
(a.out (file.o))           I               I           I
-----------------          I               I           I
        I                  I               I           I
        I                  I               I           I
        I                  I               I           I
        -----------------------------------------------
                           I
                           I
                           I
                      ------------
                      gebundener
                      Objektmodul
                      (a.out)
                      ------------
```

Bild 1-1: Verbinden von PLZ/SYS- und PLZ/ASM-Quellmodulen

2. PLZ/SYS unter WEGA

2.1. PLZ-Treiber (plz)

PLZ/SYS-Quellprogramme, die unter WEGA laufen sollen, koennen mit dem PLZ-Treiber "plz" uebersetzt und gebunden werden.
"plz" ist ein Treiber zum automatischen Aufruf des C-Preprozessors (cpp), des Compilers (plzsys), des Kodegenerators (plzcg), von "uimage", des Assemblers (as) sowie des Linkers (ld). Dabei koennen verschiedene Optionen angegeben werden. Es wird ein Objektmodul erstellt, der unter WEGA geladen und abgearbeitet werden kann.
Der PLZ-Treiber "plz" arbeitet aehnlich zu "cc" fuer C-Programme. Um ein PLZ-Quellprogramm zu uebersetzen, muss nur ein einfaches Kommando eingegeben werden um ein unter WEGA ladbares Programm zu erhalten.

Beispiel:
Ein Programm besteht aus drei PLZ/SYS-Modulen (a.p, b.p, c.p) und zwei PLZ/ASM-Modulen (d.s, e.s). Es wird uebersetzt mittels:

```
% plz a.p b.p c.p d.s e.s -o program
```

Das gebundene ausfuehrbare Programm heisst dann "program" (allgemein "a.out").
Die genaue Beschreibung zu "plz" ist zu finden in:
Kommando plz(1),
Band "WEGA-Software, Programmierhandbuch"
der P8000-Dokumentation

2.2. Einschraenkungen

Die PLZ-Programme, die mit dem PLZ-Treiber (plz) erstellt wurden, koennen keine Z-Kode-Module enthalten. Dies ergibt sich, weil der WEGA-Linker nicht die passenden Tabellen erstellen kann, um Z-Kode zu verbinden (s. Kommando ld(1), Band "WEGA-Software, Programmierhandbuch" der P8000-Dokumentation).

2.3. Vereinbarungen zur Lauffaehigkeit von PLZ/SYS-Programmen

PLZ/SYS-Programme, die unter WEGA laufen sollen, muessen einen Eintrittspunkt mit dem Namen "main" haben. Die Vereinbarung fuer "main" ist folgende:

```
global
main procedure (argc integer, argv ^^byte)
     returns (retcd integer)
```

wobei "argc" die Anzahl der Argumente ist, die von WEGA uebergeben werden und "argv" ist ein Zeiger auf ein Feld von Zeigern, einer fuer jedes Argument. Der Rueckgabeparameter "retcd" ist Null bei normaler Programmbeendigung, ansonsten wurde das Programm mit Fehler verlassen.
WEGA-Systemaufrufe (system calls) werden unterstuetzt und koennen von PLZ/SYS- Programmen aufgerufen werden. Die Bibliothek "/usr/lib/libp.a" enthaelt eine WEGA-Implementation des PLZ-I/O-Paketes, wie es von PLZ/SYS

unter dem Betriebssystem UDOS her bekannt ist (s. Abschn. 5) und eine PLZ-aufrufbare Version der Systemaufrufe.
Es gibt jedoch einige Einschraenkungen:
- die variable Anzahl von Argumenten von "exec" (execl, execle usw.) wird nicht unterstuetzt
- der "exit"-Systemaufruf wurde in "Exit" umbenannt, um ihn von dem PLZ-"exit"-Befehl zu unterscheiden
- der "signal"-Systemaufruf erfordert Funktionsparameter, die PLZ/SYS nicht erlaubt; deshalb kann der "signal"-Systemaufruf nicht aufgerufen werden.

2.4. Nutzung externer Nicht-PLZ/SYS-Prozeduren

Die vorliegende PLZ/SYS-Compiler-Version ist an die U8000-Aufrufvereinbarungen angepasst (s. U8000-Aufrufvereinbarungen, Band "WEGA-Software, Dienstprogramme" der P8000-Dokumentation). Dadurch ist es moeglich:
- C-Funktionen als externe Prozeduren zu deklarieren und sie in PLZ/SYS-Programmen direkt aufzurufen
- Bibliotheken (z.B. /lib/libc.a - Standardbibliothek) in verschiedenen Programmiersprachen zu nutzen (C, PLZ/SYS, PASCAL)
- PLZ/ASM-Prozeduren, die entsprechend den Aufrufvereinbarungen geschrieben sind, in C, PLZ/SYS bzw. PASCAL-Programmen aufzurufen

Zur Kommunikation mit dem WEGA-System (Ein-/ Ausgabe usw.) koennen damit die entsprechenden Bibliotheksfunktionen aus der C-Standardbibliothek (/lib/libc.a) genutzt werden.

3. PLZ/SYS-Compiler (plzsys)

3.1. Uebersicht

Der PLZ/SYS-Compiler uebersetzt Quellkodemodule in Zwischenkode.
Ein WEGA-Editor wird verwendet, um PLZ/SYS-Quellmodule zu erstellen. Dabei muss der Name der Quelldatei mit .p enden.
Mit der "-l"-Option erstellt der PLZ/SYS-Compiler eine Listendatei mit dem Namen der Quelldatei und der Endung .l anstelle von .p und eine Objektdatei mit der Dateinamensendung .z. Beim Erstellen der Objektdatei wird durch "plzsys" eine temporaere Datei angelegt, die nach Beendigung der Compilierung wieder geloescht wird. Die Listendatei enthaelt den Quellkode mit Zeilennummern, Befehlsnummern und Syntax-Fehlerausschriften. Die Fehlerausschrift enthaelt einen Zeiger zu jeder fehlerhaften Stelle, gefolgt von einer Fehlernummer zu jedem Zeiger. Die plzsys-Fehlerliste kann genutzt werden, um die entsprechende Fehlernummer zu interpretieren. Gelegentlich zeigt der Zeiger nicht direkt auf die fehlerhafte Stelle.
Fehlerausschriften koennen mit der "-e"-Option auch in eine separate Datei geschrieben werden.
Die Objektdatei enthaelt Z-Kode. Der Kodegenerator (plzcg) uebersetzt Z-Kode in U8000-Maschinenkode.

3.2. Aufruf plzsys

In der folgenden Beschreibung wird das Wort "filename" als ein gewoehnlicher WEGA-Pfadname verwendet.
Der Compiler wird durch die folgende shell-Kommandozeile aufgerufen. Die eckigen Klammern duerfen nicht geschrieben werden. Sie zeigen an, dass die Optionen nicht erforderlich sind.

```
% plzsys [option] filename
```

wobei "filename" die Quelle fuer einen einfachen PLZ-Modul enthaelt. Die Endung .p kann, muss aber nicht angegeben werden. Wenn sie nicht vorhanden ist, haengt sie der Compiler automatisch an den Dateinamen an, bevor die Datei eroeffnet wird. Die Optionen koennen in beliebiger Reihenfolge auftreten, getrennt durch ein Trennzeichen.

Option	Funktion
-l	erzeugt eine Listendatei mit der Endung .l; ansonsten wird keine Listendatei erzeugt
-o filename2	die Objektdatei erhaelt den Namen filename2 anstelle des Quelldateinamens mit der Endung .z; wenn die Objektdatei nicht benoetigt wird, ist fuer filename2 /dev/null anzugeben
-e	kopiert die Fehlerauschriften zu der Datei, deren Namen der Name der Quelldatei mit der Endung .e ist; wenn keine Fehler auftraten, wird die Fehlerdatei geloescht
-nd	keine Erzeugung von Symbol-, Typ-, Konstanten- und Anweisungsnummerninfomationen fuer einen hypothetischen Debugger; ansonsten werden diese Informationen erzeugt
-nc	keine Erzeugung von Debugger-Symbol-Informationen fuer CONSTANT-Namen (durch Angabe von -nd und -nc werden der Z-Kode-Umfang und die Uebersetzungszeit verkuerzt)
-t u80	generiert Z-Kode, verwendbar fuer den U880; die plzsys-Erweiterungen, wie long-Variable und Strukturvergleiche und -zuweisungen sind nicht erlaubt
-t u8000s	generiert Z-Kode, verwendbar fuer den segmentierten U8000; behandelt Zeiger als 4-Byte-Objekte anstelle von 2-Byte-Objekten; legt WORD-Daten auf gerade Adressen; erlaubt long-Variable und Strukturvergleiche und -zuweisungen

Option	Funktion
-t u8000ns	generiert Z-Kode, verwendbar fuer den nichtsegmentierten U8000; erlaubt long-Variable und Strukturvergleiche und -zuweisungen; falls -t nicht angegben ist, wird -t u8000ns angenommen (Standard)

3.3 PLZ/SYS Besonderheiten und Einschraenkungen

3.3.1. Zeichenvereinbarungen

Der PLZ/SYS-Compiler benutzt den Standard-ASCII-Zeichensatz. Klein- und Grossbuchstaben sind zulaessig, werden aber als verschiedene Zeichen gewertet. Die gemischte Schreibweise von Schluesselwoertern ist nicht zulaessig, z.B. GLOBAL und global sind zulaessig, aber nicht Global. Hexadezimalzahlen und Zeichenkettensonderzeichen koennen gross oder klein geschrieben werden, z.B. %3AbC, '%r', '%R'.

3.3.2. Zeichenketten- und Markenlaengen

Die Zeichenkettenlaengen muessen im Bereich 1-255 liegen. Marken koennen jede Laenge kleiner als 256 Zeichen haben, jedoch werden nur die ersten 127 Zeichen zur Markenidentifizierung verwendet.

3.3.3. Laenge der Quellzeile

Quellzeilen mit mehr als 120 Zeichen werden akzeptiert, aber sie werden in der Listendatei gekuerzt. Die vollstaendige Listenzeile, die noch die Zeilennummer und die Befehlsnummer enthaelt, kann bis zu 132 Zeichen lang sein. Kommentare und aufgeteilte Zeichenketten koennen ueber mehrere Zeilen gehen, die Begrenzungszeichen ! bzw. ' duerfen aber nicht vergessen werden.

3.3.4. Beschraenkung der Prozedur-, Daten- und Programmgroesse

Eine einfache Prozedur darf nicht laenger als 1000H-Byte Z-Kode sein.
Die Daten- und Programmadressierung ist innerhalb eines Moduls auf eine 16-Bit-Groesse beschraenkt. Deshalb kann ein Modul nicht mehr als 65536 Byte Daten oder Z-Kode enthalten.

3.3.5. Fehlererkennung

Nachdem der Compiler einen Fehler bemerkt hat, sucht er eine Stelle, an der die Uebersetzung fortgesetzt werden kann. In Deklarationen (ausser Prozeduren) ist das das Auftreten der naechsten Deklarationsklassenbezeichnung (CONSTANT, TYPE, GLOBAL, EXTERNAL, INTERNAL), in einer

Prozedur das naechste Schluesselwort, das den Beginn einer neuen Anweisung anzeigt (IF, FI, EXIT, REPEAT, RETURN, END usw.). Die uebersprungenen Programmteile werden nicht geprueft. Daher kann es vorkommen, dass nicht alle Fehler gleich beim ersten Compilerlauf angezeigt werden. Es sind mitunter mehrere Compilerlaeufe notwendig, bis alle Fehler erkannt und schrittweise beseitigt worden sind.
Da durch die Struktur der PLZ-Programme Programmierfehler haeufig zu Folgefehlern fuehren, ist es zweckmaessig, bei der Fehlerbeseitigung mit dem ersten Fehler zu beginnen.

3.3.6. Compilerdarstellung von konstanten Ausdruecken

Numerische Konstanten werden intern als 16-Bit-Groessen dargestellt. Jeder Operand in einem konstanten Ausdruck wird wie der Typ WORD behandelt. So ist 4/2 gleich 2 aber 4/-2 ist gleich 0, weil -2 als eine sehr grosse positive Zahl dargestellt wird.
Bei der Errechnung von konstanten Ausdruecken wird das Auftreten von Ueberlaeufen ignoriert. Wegen der Darstellung als 16-Bit-Wert darf eine Zeichenkette, die als Konstante benutzt wird, maximal 2 Zeichen lang sein.
Die Reihenfolge der Bytes bei der Abspeicherung eines Wortes ist maschinenabhaengig (beim U8000 erst der hoeherwertige Teil, dann der niederwertige Teil). Programme, bei denen diese Reihenfolge eine Rolle spielt, sind nicht ohne weiteres auf anderen Maschinen lauffaehig (z.B. beim U880 umgekehrte Abspeicherung eines 16-Bit-Wortes).

3.3.7. Zeichenkettenkonstanten oder konstante Ausdruecke

Fehler 240 tritt auf, falls eine Zeichenkettenkonstante, die groesser als 65535 ist, verwendet wird.
Konstante Ausdruecke, die waehrend des Compilerlaufes bestimmt werden muessen (wie z.B. Initialisierungswerte oder CASE-Auswahlelemente), sind beschraenkt.
Konstante Ausdruecke werden berechnet durch Verwendung von 16-Bit-Operationen mit 16-Bit-Groessen, so dass keine Fehler auftreten.
Bei Verwendung von LONG-Typen (32 Bit) muss eine Konstante oder ein konstanter Ausdruck in 32 Bit konvertiert werden. Dies wird vom Compiler wie folgt durchgefuehrt:

* Falls die Konstante oder der konstante Ausdruck LONG sein muss, dann wird die 16-Bit-Groesse in den Typ WORD uebernommen und eine WORD-in-LONG-Umwandlung durchgefuehrt (das Wort wird rechtsbuendig in ein Feld von Nullen eingeordnet)
* Falls die Konstante oder der konstante Ausdruck vom Typ LONG_INTEGER sein muss, dann wird die 16-Bit-Groesse in den Typ INTEGER uebernommen und eine INTEGER-in-LONG_INTEGER-Umwandlung durchgefuehrt (INTEGER ist vorzeichenbehaftet).

Wenn eine Konstante in einem ausfuehrbaren LONG-Ausdruck (Zuweisung oder Parameter) vorkommt, so wird die Konstante immer als eine 32-Bit-Groesse behandelt, bei der die hoechsten 16 Bit alle Null sind und alle Operationen mit der Konstanten sind volle 32-Bit-Operationen. Dies

schliesst die Negation (-) und Operationen mit anderen Konstanten ein. Anders als initialisierte Werte und CASE-Auswahlelemente werden ausfuehrbare Ausdruecke , die LONG-Operationen enthalten, waehrend der Programmabarbeitung auf der entsprechenden Maschine (U8000) bestimmt.
Konstante LONG-Ausdruecke, die waehrend der Programmabarbeitung und konstante LONG-Ausdruecke, die waehrend der Compilierung bestimmt werden, haben in vielen Faellen den gleichen Wert, so z.B., wenn der Typ LONG_INTEGER ist und der Wert im Bereich -32768 bis 32767 liegt oder wenn der Typ LONG ist und der Wert im Bereich 0 bis 65535 liegt.

Tabelle 3-1 gibt Beispiele fuer die Wertbestimmung von konstanten Ausdruecken waehrend der Compilierung bzw. waehrend der Programmabarbeitung. Ein ausfuehrbarer Ausdruck muss verwendet werden, um einen 32-Bit-Wert zu erhalten, dessen hoeherwertiges Wort weder %FFFF noch 0 ist.

Tabelle 3-1 Wertbestimmung von konstanten Ausdruecken

Konstanter Ausdruck, der waehrend der Compilierung bestimmt wird	Wert	Konstanter Ausdruck, der waehrend der Programmabarbeitung bestimmt wird	Wert
L := -1	%0000FFFF	L := -1	%FFFFFFFF
LI := -1	%FFFFFFFF	LI := -1	%FFFFFFFF
L := %FFFF	%0000FFFF	L := %FFFF	%0000FFFF
LI := %FFFF	%FFFFFFFF	LI := %FFFF	%0000FFFF
L := -%FFFF	%00000001	L := -%FFFF	%FFFF0001
LI := -%FFFF	%00000001	LI := -%FFFF	%FFFF0001
L := %AAAA*(%FFFF+1) +%BBBB	%0000BBBB	L := %AAAA*(%FFFF+1) +%BBBB	%AAAABBBB

mit L LONG
LI LONG_INTEGER
'-' unaerer '-' Operator

3.3.8. Typbestimmung von Konstanten

Der Compiler kann gewoehnlich aus dem Kontext den Typ der Konstante bestimmen (LONG oder WORD). In der Zuweisung

```
    X := 24
```

generiert der Compiler z.B. ein Wort, falls X eine 16-Bit-Groesse ist, und ein LONG-Wort, falls X eine 32-Bit-Groesse ist. Aehnlich wird der Typ von Konstanten in Parameterlisten, CASE-Ausdruecken und den meisten Relationsausdruecken bestimmt. Der einzige Fall, in dem der Compiler aus dem Kontext heraus nicht bestimmen kann, welcher Konstantentyp zu generieren ist, ist ein Relationsausdruck, wo die Konstante lexikalisch vor der Variablen auftritt (0<X).
Um die korrekte Konstante zu generieren, arbeitet der Compiler, als ob LONG-Konstanten erforderlich sind. Wenn der Compiler endlich bestimmt, welcher Typ erforderlich ist, generiert er die eigentliche Konstante. Dies hat zwei wichtige Konsequenzen:

1. Maximal 16 Konstanten koennen so korrigiert werden. Fehler 236 tritt auf, falls mehr als 16 Konstanten auftraten, bevor ihr Typ erkannt wurde
2. Falls der eigentliche Typ der Konstanten WORD ist, dann treten im Z-Kode ein oder mehrere NOP-Befehle auf. Dies verlaengert den Kode und verlangsamt die Ausfuehrung.

Um diese Probleme zu umgehen, ist die Reihenfolge der Operanden in Relationsausdruecken umzukehren (z.B. X>0 anstelle 0<X).

3.3.9. Strukturierte Rueckgabeparameter

Der Compiler erlaubt nicht die Elementauswahl aus einem RECORD-Rueckgabewert bzw. die Indizierung eines ARRAY-Rueckgabewertes.
Zum Beispiel werden bei den folgenden Prozeduren

```
    PROCA PROCEDURE RETURNS (ARRAY [10 BYTE])
    PROCR PROCEDURE RETURNS (RECORD [F1 F2 BYTE])
```

die folgenden Ausdruecke vom Compiler nicht akzeptiert:

```
    PROCA()[2]    PROCR().F1
```

Die einzigen Operationen, die in diesem Fall erlaubt sind, sind die Zuweisung und der Vergleich.
Der Compiler erlaubt den Rueckverweis eines Zeigers, der von einer Prozedur zurueckgegeben wird:

```
    EXTERNAL PROCP PROCEDURE RETURNS (^ BYTE)
    PROCP() ^
```

Wenn der Rueckgabewert eine Struktur (ARRAY, RECORD) ist, die nicht kopiert wird, kann sie durch ein Zeigerrueckgabewert ersetzt werden, der dann auf sie zurueckverweist, und dann kann eine Indizierung (ARRAY) oder Elementauswahl (RECORD) erfolgen.

```
    TYPE
      ATYPE ARRAY [S BYTE]
    EXTERNAL
      PROCA PROCEDURE RETURNS (^ATYPE)
      ...
      PROCA()^[I]
      ...
```

3.4. Compiler-Fehlerliste

Fehler	Erklaerung
	Warnungen
0	Ein Minuszeichen (-) oder ein Pluszeichen (+) wird als binaerer Operator angesehen
1	Name nicht vom nachfolgenden Ausdruck getrennt
2	Feld ohne Element
3	Keine Bereiche in einer RECORD-Deklaration
4	Falscher Prozedurname
5	Falscher Modulname
6	Konstante nicht im zulaessigen Bereich
8	Absolute Adressierung
	Fehler, die waehrend der lexikalischen Analyse gefunden wurden
10	Dezimalzahl zu gross
11	Falsches Operationszeichen
12	Falsches Zeichenkettensonderzeichen
13	Falsche Hexadezimalziffer
14	Zeichenkette mit Laenge 0
15	Falsches Zeichen
16	Hexadezimalzahl zu gross
	Schleifenfehler
20	Unpassendes OD
21	OD wird erwartet
22	Ungueltige REPEAT-Anweisung
23	Ungueltige EXIT-Anweisung
24	Ungueltige FROM-Marke
	IF-Anweisungsfehler
30	Unpassendes FI
31	FI wird erwartet
32	THEN oder CASE wird erwartet
33	Falsche Auswahlanweisung
	Erwartete Symbole
40	) wird erwartet
41	(wird erwartet
42	] wird erwartet
43	[wird erwartet
44	:= wird erwartet
45	^ wird erwartet
	Undefinierte Namen
50	Undefinierter Bezeichner
51	Undefinierter Prozedurname

Fehler	Erklaerung
	Deklarationsfehler
60	Typbezeichner wird erwartet
61	Ungueltige Moduldeklaration
62	Ungueltige Deklarationklasse
63	Ungueltige Verwendung der ARRAY[*]-Deklaration
64	Nichtinitialisierte ARRAY[*]-Deklaration
65	Ungueltiger Dimensionswert
66	Ungueltiger Feldkomponententyp
67	Ungueltige RECORD-Deklaration
68	Ungueltiger Typ bei Zeigerdeklaration
	Prozedur-Deklarationsfehler
70	Ungueltige Prozedurdeklaration
71	ENTRY wird erwartet
72	Prozedurname nach END wird erwartet
73	Formaler Parametername wird erwartet
74	Ungueltiger Typ eines formalen Parameters
	Initialisierungsfehler
80	Ungueltiger Initialisierungswert
81	Zu viele Initialisierungselemente fuer die deklarierten Variablen
82	Ungueltige Initialisierung
83	ARRAY[*] wurde durch eine leere Zeichenkette initialisiert
84	Versuch, ein nicht zu initialisierendes Datenfeld zu initialisieren
	Spezielle Fehler
90	Ungueltige Anweisung
91	Ungueltige Instruktion
92	Ungueltiger Operand
93	Operand zu gross
94	Relative Adresse ausserhalb des zulaessigen Bereiches
95	: wird erwartet
97	Doppelt vorhandener RECORD-Name
98	Doppelt vorhandene CASE-Konstante
99	Mehrfach deklarierter Bezeichner
	Ungueltige Variablen
100	Ungueltige Variable
101	Ungueltiger Operand fuer "#" oder SIZEOF
102	Ungueltiger Feldname
103	Einbeziehung einer Nicht-ARRAY-Variablen
104	Ungueltige Verwendung von "."
105	Ungueltige Verwendung von "^"

Fehler	Erklaerung
	Fehler in Ausdruecken
110	Ungueltiger arithmetischer Ausdruck
111	Ungueltiger bedingter Ausdruck
112	Ungueltiger Konstantenausdruck
113	Ungueltiger Auswahlausdruck
114	Ungueltiger Indexausdruck
115	Ungueltiger Ausdruck bei der Zuweisung
	Fehler bei Konstantengroessen
120	Konstante zu gross fuer 8 Bit
121	Konstante zu gross fuer 16 Bit
122	Konstantenfeldindex ausserhalb der Grenzen
	Fehler beim Prozeduraufruf
130	Ungueltiger arithmetischer Ausdruck
131	Ungueltiger Funktionsaufruf
132	Prozedur mit mehreren Ausgabeparametern wird erwartet
133	Zu wenige Ausgabeparameter
134	Zu viele Ausgabeparameter
135	Zu wenige Eingabeparameter
136	Zu viele Eingabeparameter
	Typinkompatibilitaeten
140	Zeichenketteninitialisierung durch eine ARRAY[*]-Deklaration, wobei der Basistyp der Komponenten ungleich 8 Bit ist
141	Typinkompatibilitaet bei Initialisierung
150	Typinkompatibilitaet in arithmetischem Ausdruck
151	Ungueltiger Operandtyp fuer Unaeroperator
152	Ungueltiger Operandtyp fuer Binaeroperator
153	Nicht zuweisbarer Typ
154	Ungueltiger Indextyp
156	Inkompatibilitaet des Parametertyps
157	Ungueltiger aktueller Parameter
158	Typinkompatibilitaet des Rueckgabeparameters
159	Rueckgabewert muss Adresse sein
160	Typinkompatibilitaet in der Zuweisung
161	Ungueltiger Operandentyp fuer Relationsoperator
162	Typinkompatibilitaet in bedingtem Ausdruck
163	Ungueltige Typumwandlung
164	Ungueltiger Relationsoperator fuer Strukturen
	Dateifehler
198	EOF wird erwartet
199	Unerwartetes EOF in Quelle gefunden - moeglicherweise befindet sich ein nicht abgeschlossenes "!" oder "'" in der Quelle

Fehler	Erklaerung
	Implementierungseinschraenkungen
230	Zeichenkette oder Bezeichner zu lang
231	Ueberlauf der Compiler-Symboltabelle
232	Prozedur zu gross
233	Linke Seite einer Zuweisung zu kompliziert
234	Zu viele Initialisierungswerte
235	Ueberlauf des Compilerstacks
236	Zu viele Konstanten im Ausdruck
237	Statischer Datenbereichsueberlauf
238	Ueberlauf des Programmbereichs
239	Zu viele INTERNAL- oder GLOBAL-Prozeduren
240	LONG-Konstanten sind nicht implementiert

Hinweis:
Fehlernummern groesser als 240 koennen auftreten. Wenn das Programm ansonsten richtig ist, weist das auf einen Fehler des Compilers hin. Es sollte dann versucht werden, die betreffenden Anweisungen anders zu formulieren.

4. Kodegenerator (plzcg)

4.1. Uebersicht

Das Ergebnis eines PLZ/SYS-Compilerlaufes ist ein Zwischenkode-Objektmodul (Z-Kode), der nicht direkt auf dem P8000 abgearbeitet werden kann. Der Z-Kode wird durch den Kodegenerator in einen Maschinenkode-Objektmodul umgewandelt.
Dieser Abschnitt beschreibt, wie der Kodegenerator zu bedienen ist. Der U8000 PLZ-Kodegenerator erwartet als Eingabe eine Z-Kode-Datei und produziert eine Datei mit verschieblichen U8000-Objektkode im zobj-Format. Dieser zobj-Modul muss dann mit "uimage" in das a.out-Format uebersetzt werden. Der von "uimage" ausgegebene Modul wird dann zusammen mit anderen Modulen im a.out-Format zu einem kompletten ausfuehrbaren Lademodul zusammengebunden.

4.2. Aufruf "plzcg"

Der Kodegenerator wird durch folgende Kommandozeile aufgerufen:

```
plzcg [-o filename2] [-s] [-l] [-v] filename1
```

wobei "filename1" die Endung .z haben kann. Die Endung .z ist optional, falls sie fehlt wird sie vom Kodegenerator automatisch angehangen, bevor die Datei eroeffnet wird.

Die Option "-o filename2" bewirkt, dass die generierte Objektdatei den Namen "filename2" erhaelt; ansonsten "t.out".
Die Option "-s" fuer geteilten Kode ist nur fuer den segmentierten U8000 zu verwenden. Die Prozeduren in einem

Modul mit geteiltem Kode koennen durch verschiedene Programme mit unabhaengig belegten Stack aufgerufen werden, ohne Aenderung des Moduls mit geteiltem Kode. Dies ist moeglich, weil die lokalen Variablen und Parameter eines Moduls mit geteiltem Kode vom aufrufenden Programm aus zugaenglich sind. Module mit nichtgeteiltem Kode enthalten Stackverbindungen im Kode, die waehrend der Ausfuehrung nicht ausgewechselt werden koennen.

Die Option "-l" erstellt eine Pseudo-Assembler-Listendatei des Moduls. Die Listendatei hat den gleichen Namen wie die Eingabedatei mit der Endung .l anstelle .z. Es wird keine Assemblerliste fuer die Daten im Modul erstellt und es gibt keine symbolischen Marken. Verbindungen zum Kode werden durch den Buchstaben "P" gekennzeichnet, lokale Daten durch "L" und globale Daten durch "G".

Die Option "-v" veranlasst "plzcg", sich beim Start zu melden und am Ende mitzuteilen, wieviel Kode und Daten erstellt wurden.

Auf dem U8000 wird die stackabhaengige Adressierung von lokalen Variablen und Parametern mit Verlust an Geschwindigkeit und Kompaktheit ausgefuehrt, so dass nur Module, die geteilt werden muessen, mit der Option "-s" fuer geteilten Kode uebersetzt werden sollten.

4.3. Kodegenerator-Fehlerliste

Wenn die Kapazitaet der internen Tabellen des Kodegenerators erschoepft ist, bricht der Kodegenerator mit einer passenden Fehlermeldung ab. Dieser Fehler kann gewoehnlich korrigiert werden durch Vergroesserung des Umfangs des unbelegten Speichers, den der Kodegenerator fuer diese Tabellen verwendet. Wenn das nicht effektiv ist, so muss die Quelle modifiziert werden, um die Tabellen zu reduzieren.

Fehler	Erklaerung
1	Unpassendes Z-Kode-Format; die Z-Kode-Datei wurde wahrscheinlich durch eine unaktuelle Version des PLZ/SYS-Compilers erstellt
2	Anweisung zu gross
3	Ausdruck zu gross
4	Prozedurschachtelung zu tief
5	Zu viele INTERNAL- oder GLOBAL-Prozeduren im Modul
6	Zu viele Alternativen in einer Auswahlanweisung
7	Prozedur zu lang

Hinweis:
Fehlernummern groesser als 7 weisen auf einen Fehler des Kodegenerators hin.

5. Umsetzung von PLZ/SYS-Modulen unter UDOS auf Betriebssystem WEGA

PLZ/SYS-Module (Version 3.0), die fuer den U880 (Betriebssystem UDOS) erstellt wurden (s. U880-PLZ/SYS, Band "UDOS-Software, Programmiersprachen" der P8000-Dokumentation), koennen i.allg. problemlos fuer den U8000 (Betriebssystem WEGA) umgesetzt werden.

Werden in dem PLZ/SYS-Modul Prozeduren aus dem UDOS PLZ-I/O- bzw. UDOS PLZ.MATH-Paket genutzt, so sind diese entsprechend durch WEGA-Bibliotheksfunktionen oder durch WEGA-Systemaufrufe oder anderweitig zu ersetzen.

Darueberhinausgehend ist in der Bibliothek "/usr/lib/libp.a" eine auf WEGA umgesetzte Version des UDOS PLZ-I/O-Paketes enthalten, die die entsprechenden Prozeduren in WEGA-Systemaufrufe umsetzt.

Folgende I/O-Prozeduren sind im WEGA PLZ-I/O-Paket enthalten:

    OPEN
    CLOSE
    GETSEQ
    PUTSEQ
    SEEK
    TRUNC
    DELETE

Diese Prozeduren sind analog zur Beschreibung des UDOS PLZ-I/O-Paketes zu verwenden. UDOS-spezifische Parameter (UNIT, REQUESTCODE) werden so umgesetzt, dass bei der Nutzung auf dem WEGA-System i.allg. keine Aenderungen notwendig sind.

Es gibt jedoch einige bisher bekannte Unterschiede:

OPEN

```
OPEN PROCEDURE (UNIT BYTE
                FILENAME_PTR ^BYTE
                FLAG BYTE)
     RETURNS   (RCODE BYTE)
```

FLAG gibt den OPEN-Typ wie folgt ab:

FLAG=0 (Eroeffnen einer existierenden Datei fuer Eingabe)
- * wenn Datei bereits existiert, wird sie aktiviert (RCODE=%80) - Dateizeiger steht am Dateianfang; Datei kann dann nur gelesen werden
- * wenn Datei nicht existiert, dann Fehler (RCODE=%C7; Datei nicht gefunden)

FLAG=1 (Eroeffnen einer Datei fuer Ausgabe)
- * wenn Datei bereits existiert, wird der Dateiinhalt geloescht (RCODE=%80)
- * wenn Datei nicht existiert, wird sie erstellt (RCODE=%80)

FLAG=2 (Eroeffnen einer existierenden Datei fuer Ausgabe)
- * wenn Datei bereits existiert, wird der Dateiinhalt geloescht (RCODE=%80) - Datei kann dann nur

```
    geschrieben werden
  * wenn Datei nicht existiert, dann Fehler (RCODE=%C6)
FLAG=3 (Eroeffnen einer Datei fuer Ein- und Ausgabe)
  * wenn Datei bereits existiert, wird sie aktiviert
    (RCODE=%80) - Dateizeiger steht am Dateianfang; nach-
    folgendes Schreiben fuehrt zum Ueberschreiben der Datei
  * wenn Datei nicht existiert, wird sie erstellt
    (RCODE=%80)
```

DELETE

```
DELETE PROCEDURE (UNIT BYTE
                  FILENAME_PTR ^BYTE)
       RETURNS   (RCODE BYTE)
```

Der Parameter "FILENAME_PTR" muss in jedem Fall angegeben werden, auch wenn die Datei bereits eroeffnet ist.

Hinweis:
Das PLZ-I/O-Paket sollte nur bei der Umsetzung von UDOS-PLZ/SYS-Programmen in WEGA-PLZ/SYS-Programme benutzt werden, da die Prozeduren UDOS-spezifische Parameter enthalten.
Bei der Neuerstellung von WEGA-PLZ/SYS-Programmen sollten die Bibliotheksfunktionen oder die Systemaufrufe benutzt werden.

U 8 0 0 0

Aufrufvereinbarungen

Inhaltsverzeichnis Seite

1. Allgemeines

Die U8OOO Aufrufvereinbarungen erlauben das Schreiben von Programmen in verschiedenen Programmiersprachen fuer den U8OOO Mikroprozessor zwecks Kommunikation mit gemeinsamen Bibliotheken. Um eine fehlerfreie Kommunikation zwischen Programmen, die in verschiedenen Programmiersprachen bzw. in U8OOO Assembler formuliert worden sind, zu gewaehrleisten, muessen drei Fragenkomplexe geklaert werden:

a) Wie werden die U8OOO-Register in einer Prozedur verwendet, und welchen Inhalt muessen die Register beim Eintritt in eine Prozedur bzw. beim Austritt aus einer Prozedur haben?

b) Wie muss der Stack vorbereitet sein beim Eintritt, bei der Ausfuehrung und beim Austritt aus einer Prozedur?

c) Wo stehen die Parameter beim Eintritt und beim Austritt aus einer Prozedur?

2. Aufteilung der U8OOO-Register

In der folgenden Abbildung ist die Aufteilung der U8OOO-Register fuer den nichtsegmentierten und den segmentierten Mode dargestellt:

```
    nichtsegmentiert                    segmentiert

        !--------!                      !--------!
    R0  !        !                      !        ! R0
        !--------!                      !--------!
        !        !                      !        !
        .        .                      .        .
        .        .  Scratch-Register    .        .
        .        .                      .        .
        !        !                      !        !
        !--------!                      !--------!
    R7  !        !                      !        ! R7
        !========!                      !========!
    R8  !        !                      !        ! R8
        !--------!                      !--------!
        !        !                      !        !
        .        .                      .        .
        .        .    Safe-Register     .        .
        .        .                      .        .
        !        !                      !        !
        !--------!                      !--------!
    R14 !        ! (wahlweise Frame-    !        ! R12/R13
        !========!  pointer)            !========!
    R15 !        !     Stackpointer     !        ! R14/R15
        !--------!                      !--------!
```

2.1. Scratch-Register

Die Scratch-Register werden benutzt:

- beim Eintritt in eine Prozedur fuer die Uebergabe von Parametern
- waehrend der Abarbeitung der Prozedur fuer beliebige Operationen
- beim Austritt aus einer Prozedur fuer die Uebergabe der Ergebnisse

2.2. Safe-Register

Der Inhalt der Register R8,...R14 im nichtsegmentierten Mode - bzw. R8,...R13 im segmentierten Mode - ist beim Eintritt in eine Prozedur bzw. Austritt aus einer Prozedur unveraendert. Das heisst, die in den genannten Registern enthaltenen Werte duerfen beim Wechsel von einer Prozedur in eine andere nicht zerstoert werden.

2.3. Stackpointer-Register

Das Register R15 im nichtsegmentierten Mode, bzw. die Register R14 (RR14) im segmentierten Mode, werden fuer Stackoperationen benoetigt.

2.4. Framepointer-Register

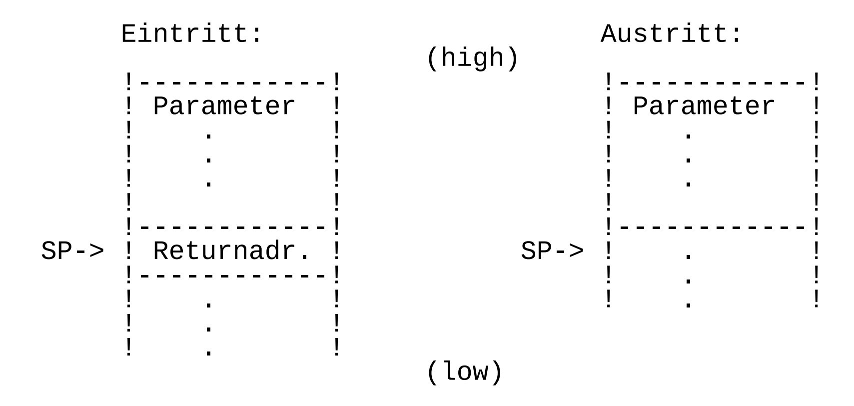
Der Framepointer verweist auf die Groesse des Stackrahmens der aktuellen Prozedur. Fuer den Framepointer wird das Register R14 im nichtsegmentierten Mode, bzw. die Register R12, R13 im segmentierten Mode verwendet (s. Abschn. 3 Stack-Organisation).

2.5. Gleitkommaregister

In Verbindung mit dem Arithmetik-Prozessor 8070 oder dem entsprechenden Software-Emulationspaket unterscheidet man bei den Gleitkommaregistern die Register FRO,...FR3 als temporaere Register und die Register FR4,...FR7 als globale Register. Die Register FRO,...FR3 werden beim Eintritt in eine Prozedur als Parameter und beim Austritt aus einer Prozedur fuer die Ergebnisse benutzt. Waehrend der Abarbeitung der Prozedur koennen die Register FRO,...FR3 fuer beliebige Operationen genutzt werden. Die Register FR4,...FR7 werden genauso gehandhabt wie die Safe-Register.

3. Stack-Organisation

Die nachfolgende Abbildung zeigt den Stackinhalt beim Eintritt in eine Prozedur und nach dem Austritt aus einer Prozedur.



Die Abbildung macht deutlich, wie der Stackinhalt auszusehen hat, wenn ein Eintritt in eine Prozedur erfolgt. An der Spitze des Stacks steht die Returnadresse. Vor der Returnadresse stehen Parameter, sofern die Scratch-Register fuer die Parameteruebergabe nicht ausreichen. Waehrend der Ausfuehrung der Prozedur befindet sich im Stack ein Datenfeld, genannt Stackrahmen oder Activation Record. Der Stackrahmen hat die zu rettenden Werte, lokale Variable und temporaere Variable zum Inhalt.
Wahlweise kann der Framepointer verwendet werden. Er ist im nichtsegmentierten Mode in R14 abgespeichert, bzw. in RR12 im segmentierten Mode. Der Framepointer verweist auf die Groesse des Stackrahmens der aktuellen Prozedur. Wird in der aktuellen Prozedur eine weitere Prozedur aufgerufen, so kann in der aufgerufenen Prozedur der Framepointer ausgewertet werden, um einen weiteren beliebigen Stackrahmen aufzubauen. Nachfolgend soll das an einem Beispiel demonstriert werden.

```
                          (high)
     !            !                 !            !
     ! Parameter  !                 ! Parameter  !
     !------------!                 !------------!
     ! Returnadr. !                 ! Returnadr. !
     !------------!                 !------------!
     ! Safe-Reg.  !                 ! alter Wert !
     !            !  Framepointer->! Framepoint.!
     !- - - - - - !                 !------------!
     ! Floating   !                 ! Safe-Reg.  !
     ! Safe-Reg.  !                 !- - - - - - !
     !------------!                 ! Floating   !
     ! lokale/    !                 ! Safe-Reg.  !
     ! temporaere !                 !------------!
     ! Variablen  !                 ! lokale/    !
     !------------!                 ! temporaere !
SP-> !     .      !                 ! Variablen  !
     !     .      !                 !------------!
     !     .      !            SP-> !     .      !
                                    !     .      !
                          (low)     !     .      !
```

Werden die Safe-Register aus programmtechnischen Gruenden in der aktuellen Prozedur benoetigt, so muessen die Werte dieser Register in den Stackrahmen gerettet werden, so dass sie vor dem Austritt aus dieser Prozedur wieder zurueckgespeichert werden koennen. Die Anordnung der Werte im Stack ist so vorzunehmen, wie sie nach einem Load-Multiple-Befehl erfolgen wuerde:

```
          sub   R15,#2*n
          LDM   @SP,Rx,#n
```

SP ist der Stackpointer (R15 im nichtsegmentierten Mode, bzw. RR14 im segmentierten Mode). Rx ist das niedrigste Register, n ist die Anzahl der Register. Fuer das Abspeichern der Safe-Register im Stack wuerde der Befehl lauten:

```
          sub  R15,#14
          LDM  @SP,R8,#7 ;im nichtsegmentierten Mode, bzw.

          sub  R15,#12
          LDM  @SP,R8,#6 ;im segmentierten Mode.
```

4. Parameter

Die Parameter werden unterteilt in Werte, Referenzen und Resultate. Wert- und Referenzparameter werden von der rufenden Prozedur an die aufgerufene Prozedur uebergeben. Die Wertparameter sind die aktuellen Parameter, die Referenzparameter sind die Adressen von den aktuellen Parametern. Die globalen Parameter von der aufgerufenen Prozedur muessen vom gleichen Typ sein, wie die aktuellen Parameter der rufenden Prozedur. Die Resultate werden ueber die formalen Parameter an die rufende Prozedur uebergeben. Jeder Parameter ist mit einem Laengenattribut versehen (length(p) fuer den Parameter p). Die Laengen fuer den Wert- und Resul-

tatparameter werden bei der Vereinbarung der formalen Parameter festgelegt. Bei Programmiersprachen, die nicht formale Parameter vereinbaren, nehmen die globalen Parameter die Laenge der aktuellen Parameter an. Die Laenge der Referenzparameter ist gleich der Laenge der Adresse (WORD fuer nichtsegmentierten Mode, LONG fuer segmentierten Mode). Die Art, der Typ und die Laenge der Parameter wird in der Programmiersprache festgelegt, in der die rufende und aufgerufene Prozedur geschrieben wurden. Der Anwender muss dies beachten, wenn ein Ruf von Prozeduren erfolgt, die in verschiedenen Sprachen formuliert worden sind.

4.1. Parameterzuordnung

Fuer die Parameteruebergabe stehen die Register R2,...R7, die Gleitkomma-Register FRO,...FR3 und der Stack zur Verfuegung. Bei den CPU-Registern unterscheidet man:

BYTE	Register (rln, rhn, n=O,...15)
WORD	Register (rn, n=O,...15)
LONG WORD	Register (rrn, n=O, 2,...,14)
QUAD WORD	Register (rqn, n=O, 4,...,12)

Die Notierung der Laenge eines Registers r lautet:

BYTE Register	Laenge (r)	= 1 Byte
WORD Register		= 2 Byte
LONG WORD Register		= 4 Byte
QUAD WORD Register		= 8 Byte

Jedes Register hat einen Satz von untergeordneten BYTE-Registern. Das untergeordnete Register von einem BYTE-Register ist das BYTE-Register selber. Die untergeordneten Register von einem WORD-Register rn sind die BYTE-Register rln, rhn. Die untergeordneten Register von einem LONG-WORD-Register rrn sind die BYTE-Register rln, rhn, rln+1, rhn+1. Die untergeordneten Register von einem QUAD-WORD-Register rqn sind die BYTE-Register rln, rhn, rln+1, rhn+1, rln+2, rhn+2, rln+3, rhn+3.

Fuer n > m gilt:

Register rxn, rn ist hoeher als Register rxm, rm.
Register rln ist hoeher als Register rhn.

Fuer die 8 Gleitkomma-Register frO,...fr7 gilt fuer n > m:

frn ist hoeher als frm.

4.2. Algorithmus der Parameteruebergabe

Wie schon erwaehnt, werden die CPU-Register R2,...R7, die Gleitkomma-Register FRO,...FR3 und der Stack fuer die Parameteruebergabe benutzt. Fuer die Parameteruebergabe gilt generell eine von links nach rechts abzuarbeitende Reihenfolge der Parameterliste.

4.2.1. Uebergabe der Wert- bzw. Referenzparameter

Besteht die Moeglichkeit, den Parameter (p) einem Register zuzuordnen, dann hat das in der Sequenz

R7,...,R2 bzw. FR3,...,FRO

zu erfolgen. Ist der Parameter ein Gleitkommawert, dann ordne p dem hoechsten verfuegbaren Gleitkomma-Register zu. Steht kein Gleitkomma-Register zur Verfuegung, dann suche das hoechste CPU-Register r heraus, das folgende Bedingungen erfuellt:

- Laenge (p) = Laenge (r)
- alle untergeordneten BYTE-Register von r muessen verfuegbar sein

Werden diese Bedingungen erfuellt, dann kann dem Parameter p das Register r zugeordnet werden. Steht keines dieser Register zur Verfuegung, dann ordne dem Parameter den naechsten verfuegbaren Speicherplatz im Stack zu. Ist erst einmal ein Parameter einem Speicherplatz im Stack zugeordnet worden, dann werden alle weitere Parameter im Stack abgespeichert.

4.2.2. Uebergabe der Resultatparameter

Wird das Ergebnis durch ein Register uebergeben, dann hat das in folgender Sequenz zu erfolgen:

R2,...,R7 bzw. FRO,...,FR3

Die Parameteruebergabe ueber die Register kann zu einer Ueberlappung fuehren, d.h., die Wert- bzw. Referenzparameter koennen von den Resultatparametern ueberschrieben werden. Bei der Uebergabe der Parameter im Stack ist eine Ueberlappung nicht moeglich. Ist p ein Gleitkommaresultat, dann weise p das naechst niedrige Gleitkommaregister zu. Steht kein Gleitkommaregister zur Verfuegung, dann suche das niedrigste CPU-Register r heraus, dass folgende Bedingungen erfuellt.

- Laenge (p) = Laenge (r)
- alle untergeordneten BYTE-Register von r muessen verfuegbar sein

Ist die Uebergabe der Resultatparameter mittels Register nicht mehr moeglich, dann speichere p in den naechsten freien Speicher im Stack ab. Ist die Laenge (p) groesser 1 (also geradzahlig) und der aktuelle Offset vom Stack ungerade, so addiere eine 1 zum aktuellen Offset vom Stack um die Laenge von p.

5. Beispiel

In diesem Beispiel wird die Parameteruebergabe anhand eines einfachen C-Programmes und eines korrespondierenden U8OOO-Programmes demonstriert.

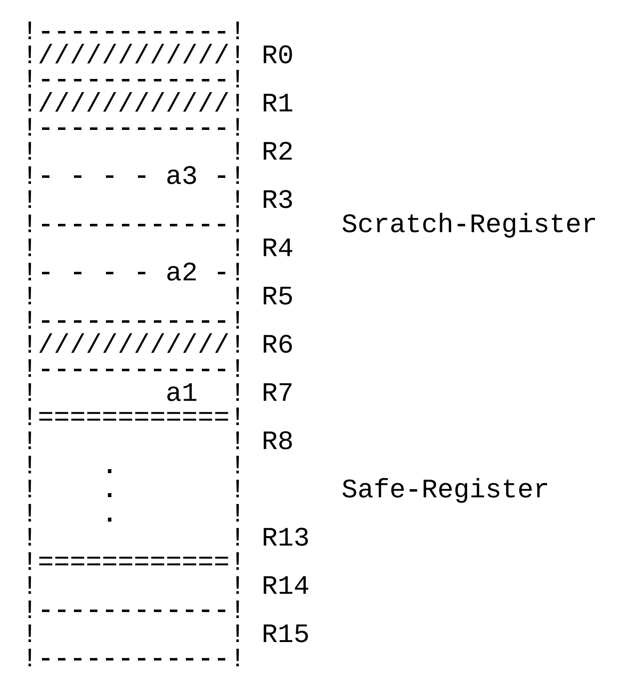
C-Programm:

```
long called (a,b,c,d,e)
long b,c;    /* Parameter b, c haben die Laenge LONG*/
int a,d,e;   /* a,d,e sind vom Typ integer          */
{
     long y;             /*   y = Resultat-Parameter */
     return(y);          /* Uebergabe des Ergebnisses*/
}
caller ()                /* rufende Prozedur */
{
     long a2,a3,x;
     int  a1,a4,a5;
     x = called(a1,a2,a3,a4,a5);
}
```

Der Aufruf 'x = called (a1,a2,a3,a4,a5)' fuehrt bei Einhaltung der Vorschrift:

- Abarbeitung der Parameterliste von links nach rechts
- Einhaltung der Sequenz R7,...,R2

zur folgenden Registerbelegung bei der Parameteruebergabe:



```
!------------!
!////////////! R0
!------------!
!////////////! R1
!------------!
!            ! R2
!- - - - a3 -!
!            ! R3
!------------!     Scratch-Register
!            ! R4
!- - - - a2 -!
!            ! R5
!------------!
!////////////! R6
!------------!
!        a1  ! R7
!============!
!            ! R8
!    .       !
!    .       !     Safe-Register
!    .       !
!            ! R13
!============!
!            ! R14
!------------!
!            ! R15
!------------!
```

Da die Register fuer die Parameteruebergabe belegt sind, muss fuer die restlichen Parameter a4, a5 der Stack zu Hilfe genommen werden.

Inhalt des Stack:

```
     (high)

     !------------!
     ! Safe-Reg./ !
     ! lokale     !
     ! Variablen  !
     !------------!      Stackrahmen von 'caller()'
     !    a5      !
     !------------!
     !    a4      ! <-SP vor Aufruf
     !============!
     ! Returnadr. !
     !------------!
     ! gerette    ! <-SP beim Eintritt in 'called()'
     ! Safe-Reg./ !
     ! lokale     !      Stackrahmen von 'called()'
     ! Variablen  !
     !------------!
     !            ! <-SP waehrend der Ausfuehrung
                         von 'called()'
     (low)
```

Die Uebergabe des Ergebnisses erfolgt entsprechend der Sequenz R2,...,R7. In diesem Beispiel steht das Ergebnis auf dem Registerpaar RR2.

Korrespondierendes U8OOO-Programm:

```
    module MODULE
    $segmented
    constant
     sp := rr14                 !Stack Pointer!
    global
    called procedure
    entry
     dec r15,#4                 !einstellen auf Anfang des !
                                !Stack-Rahmens vom called!
     ldl rr2,sp                 !ordne local Variable y dem!
                                !Return-Register zu!
     inc sp,#4
     ret
     end called

     caller procedure
     entry
     sub sp,#22                 !einstellen auf Anfang des !
                                !Stack-Rahmens vom caller!
     ld r2,sp(#18)
     ld sp,r2                   !lade a4 in das Ueberlauf-!
                                !Parameterfeld!
     ld r2,sp(#2O)
     ld sp(#2),r2               !lade a5 in das Ueberlauf-!
                                !Parameterfeld!
     ld r7,sp(#16)              !lade r7 mit a1!
     ldl rr4,sp(#4)             !lade rr4 mit a2!
     ldl rr2,sp(#8)             !lade rr2 mit a3
     call called
     ldl sp(#12),rr2            !ordne Return-Wert dem !
                                !Parameter x zu!
     add sp,#22
     ret
    end caller
```

M A K E

Programmbeschreibung

Vorwort

Diese Unterlage beschreibt das Programm make, das die Erzeugung von aktuellen Programmdateien vereinfacht. Abschnitt 1 beschreibt den Zweck und die Funktion von make. In den Abschnitten 2 und 3 werden Hinweise zur Handhabung des Programms make gegeben.
Der Anwender sollte mit dem Betriebssystem WEGA und mit der Programmierung in C oder PLZ/SYS vertraut sein.

Inhaltsverzeichnis Seite

1. Einfuehrung

1.1. Benutzung von make

In einem Projekt werden im allgemeinen grosse Programme in kleinere, verarbeitungsfreundlichere zerlegt. Unglueck-licherweise vergisst der Programmierer schnell, welche Dateien voneinander abhaengen, welche Dateien zuletzt abge-aendert wurden, oder die genaue Folge von Operationen, um eine neue Programmversion zu erstellen.
Nach langer Editorbenutzung kann man sehr leicht den roten Faden verlieren oder weiss nicht mehr, welche Dateien ver-aendert wurden und welche Objektmodule noch gelten, da eine Deklarationsaenderung das Unbrauchbarwerden anderer Dateien zur Folge haben kann. Vergisst man eine Routine zu kompi-lieren, die veraendert wurde, oder welche veraenderte Deklarationen benutzt, so fuehrt dies wohl oder uebel zum Programmstop, und Sie werden eine Menge dran zu knabbern haben, um den Fehler aufzufinden. Andererseits ist es doch sehr verschwenderisch, alles zu kompilieren, nur um auf Nummer sicher zu gehen.

Die Benutzung des Programmes make ist eine einfache Methode, Versionen von Programmen aufzustellen, die aus vielen Operationen mit zahlreichen Dateien bestehen. Wenn die Informationen ueber Dateiabhaengigkeiten und Befehls-folgen in einer speziellen Datei gespeichert sind, so reicht der einfache Befehl

```
make
```

aus, um die entsprechenden Dateien zu erneuern, unabhaengig davon, wieviel Dateien seit dem letzten make editiert wurden. Die Beschreibungsdatei ist einfach zu schreiben. Sie werden zustimmen, dass es einfacher ist, make einzuge-ben anstatt jeden einzelnen Operator. Folgendes Schema ver-deutlicht die Programmaufstellung:

Denken - Editieren - make - Pruefen

Der Befehl make schafft die fraglichen Datei einfach, korrekt und mit einem Minimum an Aufwand. Er ermoeglicht einfache Makrosubstitution und schliesst Befehle in einer einzelnen Datei ein, was bequeme Handhabung verspricht.
make ist besonders brauchbar fuer mittelgrosse Programmpro-jekte, es loest allerdings nicht die Probleme des Erstel-lens von Mehrfachquellversionen oder von Programmen groesseren Ausmasses.

2. Grundlagen

2.1. Programmfunktion

Die Hauptfunktion von make ist, den Namen fuer die Zieldatei zu finden, sie zu erneuern und zu garantieren, dass auch alle Dateien, von denen sie abhaengt, korrigiert sind. Er erstellt die Zieldatei im Falle, dass sie seit der letzten Modifikation der von ihr abhaengigen Dateien nicht modifiziert wurde.
make realisiert die Suche nach den Abhaengigkeiten bis ins Kleinste. Das Wirken der Befehle haengt von der Zeit ab, wann eine Datei zuletzt modifiziert wurde.

2.2. Programmbeispiel

Zur Veranschaulichung soll folgendes Beispiel dienen. Ein Programm prog sei herzustellen, indem drei Datein, die in der Programmiersprache C geschrieben sind: x.c, y.c und z.c zu kompilieren und anschliessend mit der Bibliothek lc zu laden sind. Angenommen, die Dateien x.c und y.c haben einige Deklarationen gemeinsam, die in der Datei defs stehen, z.c habe sie aber nicht. Dies bedeutet, dass sowohl in der Datei x.c als auch in der Datei y.c die Zeile

```
#include "defs"
```

steht. Der folgende Text beschreibt die Beziehungen und Operationen, mit denen das Programm make das Programm prog herstellen kann oder dessen neueste Version, falls in den vier Quelldateien x.c, y.c, z.c und defs geaendert wurde.

```
prog : x.o  y.o  z.o
       cc  x.o  y.o  z.o   -lc  -o  prog

x.o  y.o :   defs
```

Wenn diese Information in einer Datei namens makefile gespeichert ist, so steht der Befehl

```
make
```

fuer die Operationen, die benoetigt werden um nach einer Veraenderung der Quelldateien x.c, y.c, z.c oder defs das Programm prog zu erneuern. Make arbeitet mit drei Informationsquellen: einer vom Nutzer geliefertes Beschreibungsdatei (oben makefile genannt), die Namen der Dateien, sowie deren Datum und Uhrzeit ihrer Erstellung bzw. ihrer letzten Veraenderung, schliesslich drittens noch mit einigen internen Regeln, um fehlende Angaben in der Beschreibungsdatei sinnvoll ergaenzen zu koennen. In unserem Beispiel oben bedeutet die erste Zeile, dass prog von drei Objektdateien (.o-Dateien) abhaengt.
Wenn von diesen drei Dateien die neuesten Versionen vorhanden sind, beschreibt die zweite Zeile, wie sie zusammen mit der Bibliothek zu laden sind, damit prog hergestellt wird. Die dritte Zeile besagt, dass die Dateien x.o und y.o von

der Datei defs abhaengen. Aus dem Dateisystem von WEGA erkennt make, dass es drei gleichartig benannte C-Quelldateien (.c-Dateien) gibt, die mit den Objektdateien in Zusammenhang stehen. Mit dieser Erkenntnis und den internen Regeln ist make bekannt, wie aus den Quelldateien die Objektdateien zu erzeugen sind (indem naemlich ein Kommando cc -c zu aktivieren ist).
Die folgende ausfuehrliche Beschreibungsdatei ist zu der vorigen aequivalent. Es nutzt nicht die Intelligenz von make aus.

```
prog  :  x.o  y.o  z.o
           cc  x.o  y.o  z.o  -lc  -o  prog

x.o  :  x.c  defs
          cc  -c  x.c
y.o  :  y.c  defs
          cc  -c  y.c
z.o  :  z.c  defs
          cc  -c  z.c
```

Wenn keine der im Beispiel angegebenen Quell- oder Objektdateien veraendert wurde, seitdem prog das letzte Mal hergestellt wurde, dann sind alle Dateien aktuell, d.h., sie sind auf dem neuesten Stand, und das Kommando

```
make
```

wuerde diese Tatsache feststellen und anhalten. Wenn die Datei defs editiert wurde, werden mit Hilfe von make die Dateien x.c und y.c erneut kompiliert (nicht aber z.c) und dann wuerde prog erneut aus den aktuellen neuen Objektdateien hergestellt werden. Wenn in einem anderen Fall nur die Datei y.c veraendert wurde, dann muss nur diese erneut kompiliert und prog anschliessend geladen werden.
Wird kein Ziel angegeben, so wird das in der Beschreibungsdatei zuerstgenannte Ziel hergestellt. Bezogen auf das vorige Beispiel sorgt das Kommando

```
make x.o
```

dafuer, dass x.o erneut kompiliert wird, falls x.c oder defs modifiziert wurden.

2.3. Dateigenerierung und Makrosubstitution

Es ist haeufig nuetzlich , Regeln mit mnemotechnischen Namen, die Kommandos enthalten, zu bilden, die aber keine Datei mit dem angegebenen Namen herstellen. Vielmehr wird damit die Faehigkeit von make in vorteilhafter Weise genutzt, Dateien zu generieren und Substitutionen von Makros auszufuehren. So kann z.B. eine Eintragung save vorgesehen werden, um eine bestimmte Menge von Dateien zu kopieren. Eine Eintragung cleanup kann verwendet werden, um nicht weiter benoetigte Dateien zu loeschen. In einigen anderen Faellen kann man eine Datei mit der Laenge Null warten, um lediglich herauszubekommen, ob nach einem bestimmten Zeit-

punkt eine Aktion ausgefuehrt wurde. Dies ist besonders bei der Wartung von Archiven und Listen, die sich nicht am Ort befinden, hilfreich.
make besitzt einen einfachen Makromechanismus fuer das Substituieren von Zeilen, die die Abhaengigkeiten zwischen den Dateien beschreiben, sowie von Kommandos. Die Makros werden definiert, indem bei Kommandoargumenten oder in den Zeilen der Beschreibungsdatei ein Gleichheitszeichen angegeben wird.
Makros werden aufgerufen, indem dem Namen des Makros das Zeichen $ vorangestellt wird. Dabei muessen Namen, die laenger als ein Zeichen sind, eingeklammert werden. Beispiele fuer richtige Makroaufrufe sind :

```
$(CFLAGS)
$2
$(XY)
$Z
$(Z)
${Z}
```

Die drei letzten Aufrufe sind identisch. Allen diesen Makros werden Werte, wie nachfolgend noch gezeigt wird, waehrend der Eingabe zugewiesen. Die folgenden vier speziellen Makros veraendern Werte waehrend der Kommandoausfuehrung: $* , $@ , $? und $< (siehe Abschn. 2.4.).

```
OBJECTS = x.o y.o z.o
LIBES = -lc
prog : $(OBJECTS)
       cc $(OBJECTS)  $(LIBES)  -o prog
. . .
```

Mit dem Kommando

```
make
```

wuerde man wie frueher die drei Objektdateien zusammen mit der Bibliothek lc laden. Mit dem Kommando

```
make "LIBES= -lm -lc"
```

wuerden sie mit den zwei Bibliotheken: der mathematischen Bibliothek (-lm), und der Standardbibliothek (-lc) geladen werden, da die Makrodefinitionen in der Kommandozeile die Makrodefinitionen der Beschreibungsdatei unwirksam machen.
In WEGA-Kommandos ist es notwendig, Argumente mit Leerzeichen in Anfuehrungszeichen einzuschliessen.

2.4. Beschreibungsdateien

Eine Beschreibungsdatei enthaelt drei Arten von Informationen: Makrodefinitionen, Abhaengigkeitsinformationen und auszufuehrende Kommandos.
Eine Makrodefinition ist eine Zeile, die ein Gleichheitszeichen enthaelt, dem aber kein Doppelpunkt oder Tabulator voransteht. Dem Namen (eine Kette von Buchstaben und/oder

Ziffern), der links vom Gleichheitszeichen steht (dazwischenstehende Leerzeichen und Tabs sind bedeutungslos) wird die Zeichenkette, die dem Gleichheitszeichen folgt (fuehrende Gleichheitszeichen und Tabs sind ohne Bedeutung), zugewiesen. Richtige Makrodefinitionen sind:

```
2 = xyz
abc = -lm -lmp -lc
LIBES =
```

Bei der letzten Definition wird die leere Zeichenkette zugewiesen. Ein Makro, welches nirgends explizit definiert wurde, hat als Wert die leere Zeichenkette.
Makrodefinitionen koennen auch in der make-Kommandozeile stehen (siehe Abschn. 3.1.).
In den anderen Zeilen der Beschreibungsdatei stehen Informationen ueber die Zieldateien. Die allgemeine Form ist :

```
Ziel1[Ziel2...]:[:] [Abhaengigkeit1...] [:Kommandos] [#...]
[(tab)Kommandos] [#...]
 . . .
```

Die Angaben in Klammern koennen weggelassen werden. Die Ziele und Abhaengigkeiten werden durch Ketten von Buchstaben, Ziffern, Punkten und Schraegstrichen dargestellt (die Metazeichen von Shell * und ? werden expandiert). Ein Kommando ist irgendeine Zeichenkette, die nicht das Doppelkreuz (#), ausser in Anfuehrungsstrichen, oder Newline enthaelt. Kommandos koennen entweder nach einem Semikolon auf den Zeilen der Abhaengigkeiten oder auf einer mit Tab beginnenden Zeile, die unmittelbar der Abhaengigkeitszeile folgt, stehen.
Beginnt eine Zeile mit einem #, so werden alle dem # folgenden Zeichen ignoriert, auch Leerzeichen. Wenn eine Nichtkommentar-Zeile zu lang ist, kann sie durch \ fortgesetzt werden. Ist das letzte Zeichen in einer Zeile ein \, so werden \, Newline und nachfolgende Leer- und Tabulatorzeichen durch ein einzelnes Leerzeichen ersetzt.

Eine Abhaengigkeitszeile kann einen oder zwei Doppelpunkte enthalten. Der Name eines Zielfeldes kann in mehr als einer Abhaengigkeitszeile auftreten, aber alle diese Zeilen muessen den Doppelpunkt gleichartig, d.h. einfach oder doppelt, verwenden. Damit hat es folgende Bewandnis:

- Im Normalfall des einfachen Doppelpunktes darf hoechstens eine dieser Abhaengigkeitszeilen eine Folge von Kommandos enthalten. Wenn das Ziel veraltet ist infolge irgendwelcher Abhaengigkeiten, die auf irgendwelchen dieser Zeilen stehen, und eine Folge von Kommandos spezifiziert wurde (vielleicht sogar nur eine leere, die einem Semikolon oder Tab folgt), wird sie ausgefuehrt. Ist dies nicht der Fall, wird die Standarderzeugungsregel aufgerufen.

- Im Fall des doppelten Doppelpunktes kann jeder Abhaengigkeitszeile eine Folge von Kommandos zugeordnet werden. Wenn ein Ziel veraltet ist, wegen irgendeiner

der Dateien auf einer so bestimmten Zeile, werden die zugehoerigen Kommandos ausgefuehrt. Eine interne Regel kann ebenfalls ausgefuehrt werden. Diese ausfuehrliche Form ist insbesondere bei der Bearbeitung von Archivdateien wertvoll.

Wenn ein Ziel hergestellt werden muss, wird die Folge der Kommandos abgearbeitet. Normalerweise wird jede Kommandozeile ausgegeben und wird dann einzeln, nachdem die Makros substituiert wurden, an Shell uebergeben (das Ausgeben wird im Schweigemodus unterdrueckt, oder auch, wenn die Zeile mit dem Zeichen @ beginnt). make beendet die Abarbeitung, wenn ein Kommando einen Fehler signalisiert. Die Fehler werden ignoriert, wenn auf der Kommandozeile fuer make das Flag "-i" angegeben wurde. Die Fehler werden auch ignoriert, wenn der scheinbare Zielname .IGNORE in der Beschreibungsdatei steht oder wenn die Kette von Kommandos in der Beschreibungsdatei mit einem Bindestrich beginnt. Einige Kommandos von WEGA uebergeben einen bedeutungslosen Status. Weil jedes Kommando fuer sich an Shell uebergeben wird, muss man bei bestimmten Kommandos aufpassen (z.B. bei cd und anderen speziellen Shell-Kommandos), die nur eine Bedeutung innerhalb eines Shell-Prozesses haben. Die Ergebnisse sind bei Ausfuehrung der naechsten Zeile vergessen.

Vor der Ausfuehrung jedes Kommandos werden bestimmte Makros automatisch eingestellt. $@ wird auf den Namen der Datei gesetzt, die hergestellt werden soll. $? stellt die Kette von Namen dar, die juenger als das Ziel sind. Wenn das Kommando mittels einer impliziten Regel (siehe Abschn. 3.2.) generiert wurde, dann ist $< der Name der Datei, die die Aktion verursachte und $* ist das Praefix, das die aktuellen und die abhaengigen Dateinamen gemeinsam haben.
Wenn eine Datei herzustellen ist, es aber keine expliziten Kommandos oder relevanten internen Regeln gibt, dann werden die Kommandos, die mit dem Namen .DEFAULT verknuepft sind, verwendet. Wenn es solch einen Namen nicht gibt, wird von make eine Meldung ausgegeben, und die Abarbeitung wird gestoppt.
Ziele und Abhaengigkeiten sind gebraeuchliche Dateinamen. Es existiert eine spezielle Schreibweise fuer Ziele oder Abhaengigkeiten in Archiven ar(1).

Die Schreibweise

```
          archive(file)
oder
          archive((entry point))
```

bezieht sich auf die Datei im Archiv. Modifikationsdaten werden auf im Archiv gespeicherte Daten zurueckgefuehrt. So bezieht sich z.B.

```
          libc.a(printf.o)
oder
          libc.a((_printf))
```

auf den Objektmodul printf.o im Archiv libc.a .

3. Befehlshandhabung

3.1. Argumente

Das Kommando make kann vier Arten von Argumenten benutzen. Dies sind Makrodefinitionen, Flags, Beschreibungsdateien und Zieldateinamen. Die Syntax fuer das Kommando, d.h. die fuer die Kommandozeile, ist:

```
make [ Flags ] [ Makrodefinitionen ] [ Ziele ]
```

Die folgende Zusammenstellung der Kommandooperationen zeigt, wie die Argumente zu verstehen sind.

Zuerst werden alle Argumente, die Makrodefinitionen sind (dies sind solche Argumente, die ein Gleichheitszeichen enthalten) analysiert und die Zuweisung wird ausgefuehrt. Die in der Kommandozeile angegebenen Makros machen die in der Beschreibungsdatei stehenden gleichnamigen Makrodefinitionen ungueltig.

Als naechstes werden dann die Flags abgefragt. Die moeglichen Flags sind:

-i Ignoriere den vom aufgerufenen Kommando uebergebenen Fehlerkode. Dieser Modus besteht, wenn der scheinbare Zielname .IGNORE in der Beschreibungsdatei steht.

-s Schweigemodus. Die auszufuehrenden Kommandos werden nicht ausgegeben. Dieser Modus besteht auch , wenn der scheinbare Zielname .SILENT in der Beschreibungsdatei steht.

-r Die internen Regeln sind nicht zu verwenden.

-n Modus NICHT AUSFUEHREN. Die Kommandos werden ausgegeben aber nicht ausgefuehrt.

-t BERUEHRE (touch) die Zieldatei (dies hat zur Folge, dass die Angaben ueber den Zeitpunkt der Erstellung bzw. der letzten Modifikation durch die gerade aktuelle Zeit ersetzt werden). Die Kommandos selbst werden nicht ausgefuehrt.

-q FRAGE (question). Das Kommando make uebergibt als Status Null, wenn die Zieldatei auf dem neuesten Stand ist, und einen von Null verschiedenen Wert, wenn die Zieldatei veraltet ist.

-p (print). Gibt alle Makrodefinitionen und Zielbeschreibungen aus.

-d Pruefmodus (debug). Es werden detaillierte Informationen ueber die untersuchten Dateien und deren Zeitpunkte ausgegeben.

-f Beschreibungsdateiname. Das diesem Flag folgende Argument wird als der Name der Beschreibungsdatei angese-

hen. Mit dem Dateinamen "-" wird die Standardeingabe angegeben. Ist kein Flag "-f" angegeben, so wird als Beschreibungsdatei eine Datei mit dem Namen makefile oder Makefile aus dem aktuellen Directory verwendet. Der Inhalt der Beschreibungsdateien setzt eventuell vorhandene interne Regeln ausser Kraft.

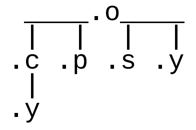
Abschliessend wird von den uebrigen Argumenten in der Kommandozeile angenommen , dass sie Namen von herzustellenden Zielen sind. Sie werden in der Reihenfolge von links nach rechts bearbeitet. Wenn es keine solchen Argumente gibt, wird der erste Name aus der Beschreibungsdatei, der nicht mit einem Punkt beginnt, als herzustellendes Ziel angesehen.

3.2. Implizite Regeln

Das make-Programm benutzt eine Tabelle der in WEGA ueblichen Suffixe und eine Menge von Transformationsregeln, die die standardmaessigen Abhaengigkeiten und die diesbezueglichen Kommandos bereitstellen.
Es folgt die Liste der standardmaessigen Suffixe :

```
.o   Objektdatei
.c   C-Quelldatei
.s   Assemblerquelldatei
.y   YACC-C-Quelldatei
.p   PLZ/SYS-Quelldatei
```

Das folgende Diagramm veranschaulicht die standardmaessigen Transformationspfade. Wenn es zwei Pfade gibt, die ein Paar von Suffixen verbinden, so wird der laengere nur dann verwendet, wenn die Zieldatei vorhanden ist oder in der Beschreibungsdatei genannt wurde.



Wenn z.B. die Datei x.o benoetigt wird und es in der Beschreibungsdatei oder dem Directory nur eine Datei x.c gibt, so wird x.o durch Kompilation aus x.c hergestellt. Wenn es auch noch eine Datei x.y gaebe, wuerde die Grammatik x.y zuerst durch den Generator YACC bearbeitet und dessen Ergebnis anschliessend kompiliert werden. Wenn jedoch kein x.c, sondern nur x.y angegeben ist, zieht make die direkte Verbindung, wie sie im obigen Graphen angegeben ist, der Verbindung mit den dazwischenliegenden C-Sprachendatei vor.
Wenn die benutzten Makronamen bekannt sind, besteht die Moeglichkeit, die Namen von einigen der standardmaessig verwendeten Compiler oder die Flag-Argumente, mit denen sie aufgerufen werden, zu aendern. Die Compiliernamen sind die Makros AS, CC, PLZ, und YACC. Das Kommando

```
make CC=newcc
```

bewirkt, dass anstatt des ueblichen Kommandos wahlweise Flags angebbar sind. So bewirkt

```
make "CFLAGS=-O"
```

dass der optimierende C-Compiler verwendet wird.

3.3. Suffixe und Wandlungsregeln

Damit dem make-Programm bekannt ist , welche Suffixe von Interesse sind und wie eine Datei mit einem bestimmten Suffix in eine Datei mit einem anderen bestimmten Suffix zu transformieren ist, sind diese Informationen in einer internen Tabelle gespeichert. Diese Tabelle hat die Form einer Beschreibungsdatei. Wird das Argument "-r" benutzt, so wird diese Tabelle nicht verwendet.
Die Liste der Suffixe hat den Namen .SUFFIXES. Wenn MAKE eine Datei sucht, dann wird es mit jedem der Suffixe aus der Liste gesucht. Sobald eine Datei gefunden wird und es eine Wandlungsregel dafuer gibt, arbeitet make normal.
Die Wandlungsregelnamen bestehen aus der Verkettung zweier Suffixe. So ist z.B. der Name der Regel mit der eine PLZ/SYS-Quelldatei (.p) zu einer Objektdatei transformiert wurde, dementsprechend .p.o . Wenn die Regel vorhanden ist und keine explizite Kommandofolge in der Beschreibungsdatei des Nutzers angegeben wurde, dann wird die Folge von Kommandos der Regel "p.o" verwendet. Wenn ein Kommando unter Verwendung dieser durch die Suffixe bestimmten Regeln generiert wird, dann gibt das Makro $* den Wert des Stammes (alles ausser dem Suffix) des Namens von der herzustellenden Datei an. Das Makro $< gibt in diesem Fall den Namen der Abhaengigkeit an, die die Aktion verursachte.

Die Reihenfolge der Suffixe in der Liste ist signifikant. Die Liste wird von links nach rechts gemustert. Der erste Name, der gebildet wird, und fuer den sowohl eine Datei existiert, als auch eine damit in Zusammenhang stehende Regel, wird verwendet. Der Nutzer kann an die Suffixliste weitere Suffixe anfuegen, indem er in seiner Beschreibungsdatei nach .SUFFIXES diese weiteren Suffixe angibt. Steht nach .SUFFIXES nichts, so wird die aktuelle Liste geloescht. Letzteres ist notwendig, wenn die Reihenfolge der Suffixe geaendert werden soll.
Hier ein Auszug aus der Standard-Regel-Datei:

```
.SUFFIXES : .o .c .p .y .s
YACC=yacc
YACCR=yacc -r
YACCE=yacc -e
YFLAGS=
CC=cc
AS=as -u
CFLAGS=
PLZ=plz
```

```
PFLAGS=
.c.o :
     $(CC) $(CFLAGS) -c $<
.p.o :
     $(PLZ) $(PFLAGS) -c $<
.s.o :
     $(AS) -o $@ $<
.y.o :
     $(YACC) $(YFLAGS) $<
     $(CC) $(CFLAGS) -c y.tab.c
     rm y.tab.c
     mv y.tab.o $@
y.c :
     $(YACC) $(YFLAGS) $<
     mv y.tab.o $@
```

3.4. Programmbeispiel

Als Beispiel fuer die Benutzung von make wollen wir die Beschreibungsdatei vorstellen, die verwendet wird, um das Programm make selbst zu warten. Der Kode fuer make besteht aus einer Reihe von C-Quelldateien und einer YACC-Grammatik. Die Beschreibungsdatei sieht wie folgt aus:

```
# Beschreibungsdatei fuer das MAKE-Kommando

p = lpr
FILES = Makefile version.c defs main.c doname misc.c
        files.c dosys.c gram.y
OBJECTS = version.o main.o doname.o misc.o files.o
          dosys.o gram.o
LIBES = -lS
LINT = lint -p
CFLAGS = -O

make:  $(OBJECTS)
       cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
       size make

$(OBJECTS): defs

cleanup:
       -rm *.o gram.c
       -du

install:
       @size make /usr/bin/make
       cp make /usr/bin/make ; rm make
print:
       $(FILES)    # print recently changed files
       pr $? | $P
test:
       make -dp | grep -v TIME >1zap
       /usr/bin/make -dp | grep -v TIME >2zap
       diff 1zap 2zap
       rm 1zap 2zap
```

```
lint:  dosys.c doname.c files.c main.c misc.c /
       version.c gram.c
       $(LINT) dosys.c doname.c files.c main.c /
       misc.c version.c gram.c rm gram.c
```

make gibt gewoehnlich jedes Kommando aus, bevor es ausgefuehrt wird. Die folgende Ausgabe entsteht, wenn man sich im Directory befindet, das nur die Quelldateien und die Beschreibungsdatei enthaelt, und das einfache Kommando

```
make
```

eintippt:

```
cc  -c version.c
cc  -c main.c
cc  -c doname.c
cc  -c misc.c
cc  -c files.c
cc  -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  -c gram.c
cc  version.o main.o doname.o misc.o files.o
    dosys.o gram.o -ls -o make
    13188+3348+3044 = 19580b = 046174b
```

Obwohl keine der Quelldateien und keine der Grammatiken mit Namen bei den Kommandos direkt in der Beschreibungsdatei erwaehnt wird, findet sie make, indem es seine Suffix-Regeln verwendet und die benoetigten Kommandos ausgibt. Die Zahlen in der obigen Ausgabe ergeben sich durch das Kommando size make, waehrend die Ausgabe des Kommandos selbst durch ein Zeichen @ unterdrueckt wird.
Die uebrigen Eintragungen in der Beschreibungsdatei sind hilfreiche Folgen fuer die Wartung. Die Eintragung print sorgt dafuer, dass nur die Dateien ausgegeben werden, die seit dem letzten Kommando make print geaendert wurden. Eine Datei print mit der Laenge Null wird gewartet, um die Zeit der Ausgabe zu verfolgen. Das Makro $? in der Kommandozeile sammelt dann nur die Namen der Dateien auf, die veraendert wurden, seitdem print "beruehrt" wurde. Die Ausgabe kann an verschiedene Drucker gesendet werden, oder an eine Datei, indem die Definition des Makros P veraendert wird:

```
make print "P = opr -sp"
```

oder

```
make print "P = cat >zap"
```

3.5. Hinweise und Warnungen

Erfahrungsgemaess entstehen die meisten Schwierigkeiten aus dem nicht gruendlichen Durchdenken und Verstehen der von make zu behandelnden Abhaengigkeiten. Wenn eine Datei x.c eine Zeile [#include "defs"] enthaelt, haengt eben die Objektdatei x.o von defs ab, und nicht die Quelldatei x.c (wenn naemlich defs veraendert wird , ist nicht die Datei x.c zu behandeln, sondern die Datei x.o muss erneut hergestellt werden).
Um festzustellen, was make tun wuerde, ist die Option "-n" sehr hilfreich. Das Kommando

```
make -n
```

befiehlt, diejenigen Kommandos auszugeben, die make ausfuehren lassen wuerde, ohne sie in Wirklichkeit auszufuehren (also ohne die Zeitpunkte der betroffenen Dateien zu veraendern).
Wenn es sicher ist, dass eine Veraenderung einer Datei "harmlos" ist (z.B. wenn einer Datei eine neue Definition hinzugefuegt wird), kann die Option -t (touch) eine Menge Zeit sparen. Anstatt laufend zu rekompilieren, haelt make die Modifikationszeiten der fraglichen Datei auf dem neuesten Stand.

Das Kommando

```
make -ts
```

(touch silently) bewirkt, dass die relevanten Dateien aktualisiert werden. Es muss voellig klar sein, dass mit dieser Moeglichkeit sehr sorgfaeltig umgegangen werden muss, da mit dieser Operation die urspruengliche Absicht von make hintergangen wird und die betreffenden vorangegangenen zeitlichen Beziehungen zerstoert werden.
Das Testflag "-d" (debug) veranlasst make, eine sehr ausfuehrliche Beschreibung davon auszugeben, was es gerade macht, einschliesslich der Dateizeiten. Die Ausgabe ist sehr umfangreich. Diese Option ist nur als letzter Ausweg zu empfehlen.



KOMBINAT VEB
**ELEKTRO-APPARATE-WERKE
BERLIN-TREPTOW
»FRIEDRICH EBERT«**

HEIM-ELECTRIC
EXPORT-IMPORT
Volkseigener Außenhandelsbetrieb
der Deutschen Demokratischen Republik

EAW-Automatisierungstechnik Export-Import

Storkower Straße 97
Berlin, DDR - 1055
Telefon 432010 · Telex 114158 heel dd

**VEB ELEKTRO-APPARATE-WERKE BERLIN-TREPTOW
»FRIEDRICH EBERT«**
Stammbetrieb des Kombinats EAW
DDR - 1193 Berlin, Hoffmannstraße 15-26
Fernruf: 2760
Fernschreiber: 0112263 eapparate bln
Drahtwort: eapparate bln